Titre: Title:	Approche basée sur l'intelligence collective pour la génération de données de test par mutation
Auteur: Author:	Kamel Ayari
Date:	2007
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ayari, K. (2007). Approche basée sur l'intelligence collective pour la génération de données de test par mutation [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8030/

UNIVERSITÉ DE MONTRÉAL

APPROCHE BASÉE SUR L'INTELLIGENCE COLLECTIVE POUR LA GÉNÉRATION DE DONNÉES DE TEST PAR MUTATION

KAMEL AYARI
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2007

Your file Votre référence
ISBN: 978-0-494-35661-6
Our file Notre référence
ISBN: 978-0-494-35661-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

APPROCHE BASÉE SUR L'INTELLIGENCE COLLECTIVE POUR LA GÉNÉRATION DE DONNÉES DE TEST PAR MUTATION

présenté par: AYARI Kamel

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. MERLO Ettore, Ph.D., président

M. ANTONIOL Giuliano, Ph.D., membre et directeur de recherche

Mme. BOUCHENEB Hanifa, Doctorat, membre

À mes parents, Mohamed et Yamina. Vous avez ma gratitude infinie.

REMERCIEMENTS

Pour commencer je remercie Dr. Giuliano Antoniol qui m'a guidé vers l'accomplissement de ce travail. C'est grâce à sa persévérance que ce document est achevé. Je remercie grandement Salah Bouktif pour son encadrement et conseils précieux durant tout ce travail. Je tiens aussi à remercier Dr. Ettore Merlo qui, comme le fit le fameux mage avec ces disciples, m'a initié à la cuisine des recettes de réingénierie du logiciel et des techniques de compilation. Je remercie mon ami Peyman Meshkinfam. Il m'a apporté plus d'aide que ce dont il s'en doute. En fin, je n'aimerai pas oublier de remercier Dr. José Fernandez qui m'a encouragé à entamer des études supérieurs.

La liste des personnes à remercier dépasse certainement l'espace alloué à cette partie de ce document. Je remercie alors tous ceux qui m'ont aidé à accomplir ce travail.

RÉSUMÉ

Dans beaucoup d'entreprises du logiciel, le test du logiciel est responsable de plus que 40 à 50% de la totalité des coûts du développement. En outre, le test et la génération de cas de test sont parmi les activités les plus laborieuses et difficiles techniquement dans un projet de logiciel. Par conséquence, des techniques réduisant le besoin d'intervention manuelle affecteront positivement les coûts du projet. En effet, le test exhaustif est souvent impraticable en raison de l'espace d'exécution potentiellement infini ou en raison du coût élevé face à des limitations serrées du budget. D'autres techniques telles que l'inspection du code sont connues pour être plus efficaces, mais encore plus coûteuses que le test. Malheureusement, des défauts présents dans un logiciel déployé peuvent affecter la sécurité ou encore des applications à missions critiques avec des conséquences catastrophiques. Le test par mutation, à l'origine proposé par DeMillo en 1978, consiste en l'injection de simples fautes dans le programme original afin d'obtenir des versions erronées de ce programme : les mutants. Par la suite, des données de test sont produites afin d'atteindre le plus haut score de mutation possible, c.-à-d., tuer le nombre le plus élevé de mutants. Un mutant est dit tué s'il demontre un comportement différent de celui du programme à tester pour les mêmes données de test. Un ensemble de cas de test est plus adéquat qu'un autre s'il tue un nombre plus grand de mutants. D'un autre coté, un jeu de test est préféré à un autre s'il contient moins de cas de test tout en étant plus proche du critère d'adéquation. Dans ce cas-ci, le critère d'adéquation est d'avoir le plus haut score de mutation. Intuitivement, le test par mutation favorise les jeux de tests de qualité et possède un potentiel d'automatisation élevé. Dans ce mémoire, nous adressons le problème de la génération automatique des données de test étant une activité coûteuse en calcul dans le processus de test. Ce travail résume l'utilisation d'une approche d'intelligence collective afin de produire des données qui tuent des mutants dans le contexte du test par mutation. Dans la technique proposée dans ce mémoire, la génération des données de test est exprimée sous forme d'un problème de minimisation

guidé par une fonction de coût, une fonction objectif inspirée par le travail de L. Bottaci. La fonction objectif de Bottaci est définie de manière à ce qu'un cas de test est capable de tuer un mutant s'il satisfait trois conditions employées par J. Offutt dans CBT, à savoir, la condition d'atteinte, la condition nécessaire et la condition suffisante. En effet, cette fonction objectif mesure combien un cas de test est-il proche de tuer un mutant. Ce mémoire a adopté l'optimisation par colonie de fourmis (CDF) comme algorithme métaheuristique pour résoudre le problème de minimisation. Deux raisons justifient ce choix. D'abord, les métaheuristiques se sont avérés des approches adéquates pour la génération de données dans le contexte du test basé sur la couverture. Ensuite, la CDF permet la mise en application de l'approche dite "exécution intelligente" d'une manière naturelle étant doné que la CDF permet intrinsèquement une recherche parallèle. Dans notre approche, les fourmis ont la mission de tuer un mutant à la fois, en cherchant des données de test qui satisfont les trois conditions d'Offutt. En outre, notre algorithme de la CDF est étendu à l'aide d'un procédé d'estimation de la densité de probabilité qui guide automatiquement et raffine la recherche vers des régions prometteuses. Les contributions principales et les innovations de ce mémoire sont :

- Une nouvelle approche basée sur l'intelligence collective pour la génération automatique de données de test dans le contexte du test par mutation, qui réduit naturellement le coût de calcul dans une telle stratégie de test ;

- l'exploitation d'une nouvelle technique émergente de recherche, la CDF, pour faciliter la génération de données de test et comparer les résultats à l'escalade des collines à recommencement aléatoire, l'algorithme génétique et la recherche aléatoire, sur deux programmes;

- l'incorporation d'une nouvelle idée basée sur un procédé d'estimation de la densité de probabilité pour définir de manière automatique et guider la recherche vers des régions prometteuses;

- Une adaptation de la CDF au problème de la génération des données de test pour tuer des mutants.

Ce mémoire démontre que la CDF a réalisé une meilleure performance que les autres métaheuristiques que nous avons implémenté.

ABSTRACT

In many software organizations software testing accounts for more than 40-50% of the total development costs. Also, testing and test case generation are among the most labor intensive and technically difficult activities in any software project. So, techniques reducing the need for manual intervention will positively affect project costs. Indeed, thorough testing is often unfeasible because of the potentially infinite execution space or high cost with respect to tight budget limitations. Other techniques such as code inspection are known to be more effective, but even more costly than testing. Unfortunately, defects slipped into deployed software may crash safety or mission critical applications with catastrophic consequences. Mutation testing, originally proposed by DeMillo in 1978, consists in injecting simple faults into the original program in order to obtain faulty versions of the program: the mutants. Then, test data are produced to attain the highest possible mutation score, i.e., to kill the highest number of mutants. A mutant is said to be killed if it behaves in a different way than the original program for the same test data. A set of test cases is more adequate than another if it kills a larger number of mutants. On the other hand, a test suite is preferred over others if it contains fewer test cases and is closer to the adequacy criterion. In this case, the adequacy criterion is having the highest mutation score. Intuitively, mutation testing promotes high quality test suites and has high potential for automation. In this thesis, we address the problem of automatic test data generation, as it is a computational expensive activity in the testing process.

This work summarizes the use of a swarm intelligence approach in order to generate data that kills mutants in the context of mutation testing. In the technique proposed in this thesis, test data generation is mapped into a minimization problem guided by a cost function, a fitness function inspired by Bottaci work. The Bottaci fitness function is defined in a way that a test case is able to kill a mutant if it satisfies three conditions used by Offutt in CBT, namely, the reachability, the necessary and the sufficiency conditions.

Indeed, this fitness function measures how close a test case is to kill a mutant. This thesis adopted Ant Colony Optimization (ACO) as the metaheuristic algorithm to solve the minimization problem. Two reasons justify our choice. First, metaheuristics have been proven to be suitable approaches for data generation in the context of coverage based testing. Second, ACO leads to implement the well-known "do smarter" approaches in a natural way because ACO intrinsically allows a parallel search. In this thesis ants have the mission of killing one mutant each time by searching a test data that satisfies the three Offutt conditions. Furthermore, our ACO algorithm is enhanced by a probability density estimation process that automatically guides and refines the search in promising regions. This thesis main contributions and innovations are:

- A new swarm intelligence approach for automatic test data generation in the context of mutation testing, which naturally reduces the computational cost in such test strategy;
- Exploitation of a new emergent search technique, ACO, to facilitate test data generation and compare results with Hill Climbing, Genetic Algorithm and random search on two programs ;
- Incorporation of new ideas based on a probability density estimation process to automatically define and guide the search toward promising regions;
- A customization of ACO to the problem of generating test data to kill mutants.

This thesis demonstrates that ACO performed far better than the other metaheuristics we implemented.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES NOTATIONS ET DES SYMBOLES

ACO : Ant colony optimization

AG : Algorithmes génétiques

CBT : Constraint-based test data generation technique

CDF : Colonie de fourmis

CFG : Control flow graph ou Graphe de flot de contrôle

CT : Cas de test

DDR : Dynamic domain reduction

DT : Données de test

$ECRA$: Escalade des collines à recommencement aléatoire

FDP : Fonctions de densité de probabilité

GDD : Graphe de dépendance des données

JT : Jeu de tests

PAT : Programme à tester

RS : Recuit simulé

TE : Test évolutionniste

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

1.1 Le Test du Logiciel

Le test du logiciel est un processus qui vise à vérifier que le logiciel possède le niveau de qualité désiré. Pour tester un programme nous le soumettons à des entrées choisies. Le comportement observé du programme à tester (PAT) est alors enregistré pour être évalué et comparé à un comportement spécifié. Ce comportement est celui qui devrait satisfaire les requis exprimés et implicites de l'utilisateur du programme. Toute déviation par rapport à ce comportement doit être rapportée et une investigation devrait mener à l'identification des causes de cette déviation et à son élimination.

Le principe du test du logiciel est illustré par la figure 1.1.

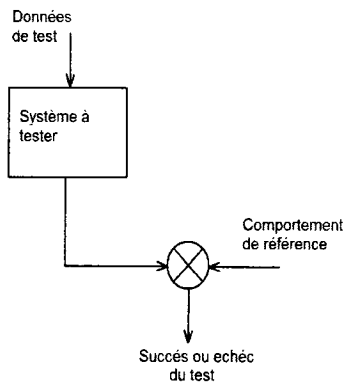


Figure 1.1 Principe du Test du Logiciel

Le test du logiciel comprend le test des unités, le test du système, le test d'intégration et le test d'acceptation (Leung and White, 1990). Ces niveaux de test découlent des

phases de développement du logiciel lui-même. En général, plus une faute est découverte tardivement, plus les coûts qu'elle engendre sont importants. Ainsi, une faute au niveau des spécifications coûtera plus cher si constatée seulement au déploiement du système.

On distingue aussi le test fonctionnel ou boite noire (Perry, 1986) du test structurel ou boite blanche (Myers, 1979).

Le test boite noire se concentre sur l'évaluation de l'exactitude de l'implémentation des fonctions requises dans le logiciel sans regarder la manière avec laquelle cela a été fait. Par exemple, pour une fonction qui retourne la surface d'un rectangle pour lequel on fournit la longueur et la largeur, le test boite noire voudra savoir si la valeur retournée par cette fonction correspond bien à la superficie du rectangle en question. Il ne s'attarde pas sur les instructions avec lesquelles le calcul de la valeur de retour se fait.

Le test boite blanche se sert de l'analyse de la structure interne du PAT pour le tester. Par exemple, le test boite blanche voudra que chaque instruction du programme soit exécutée. De cette façon on peut verifier que le programme ne contienne pas du code non exécutable ou du code mort. Certains critères du test boite blanche, telsque l'exécution de tous les chemins, peuvent être impossibles à satisfaire (Merlo and Antoniol, 1999)

Pour la fonction de calcul de la superficie d'un rectangle décrite précédemment, le test boite blanche peut viser l'exécution de toutes les instructions dans le code en vue de voir si des fonctionnalités non désirées y ont été implémentées.

1.2 Définition du Problème

1.2.1 Le Test du Logiciel et la Génération des Données de Test

En plus d'être complexe, le test du logiciel est généralement très coûteux. Le coût du test est de l'ordre de 50% du coût total du projet de développement (Pressman, 1992).

Dans le cadre d'un projet de test, le testeur est appelé à choisir les données de test (DT). Ce sont les données qui vont être utilisées pour tester le PAT. Par la suite, il doit définir le comportement normal du PAT pour ces DT. Le testeur est ainsi confronté à deux difficultés principales :

- le choix de la méthode d'obtention des DT,
- la définition du comportement de référence ou le comportement jugé 'normal'.

La sélection des DT consiste à choisir, parmi toutes les données admissibles, un ensemble réduit ayant la capacité de dévoiler les éventuelles fautes contenues dans le PAT.

On définit un cas de test (CT) comme étant l'ensemble des entrées à fournir au PAT pour une exécution auquel on ajoute les résultats attendus du PAT. Un CT est donc composé d'une série de données d'entrée et d'une série de données de sortie. L'ensemble des cas de test retenus pour tester le PAT est dit jeu de tests (JT).

On dit qu'un CT échoue si le comportement du PAT est différent de celui prévu dans le CT. Si le comportement du PAT correspond à celui décrit dans le CT on dit que le CT réussit. De la même manière, un JT échoue si au moins un de ces CT échoue. Autrement le JT réussit.

1.2.1.1 Le Test Combinatoire et Complexité de l'Espace de Recherche

À moins de couvrir tous les schémas d'exécution possibles, un JT ne peut pas garantir que le PAT qui réussit tous les tests est exempt de fautes. Or, le test exhaustif s'avère impossible pour des systèmes réels. À titre d'exemple, considérons une fonction qui prend en entrée trois valeurs entières codées sur 32 bits. Pour procéder à un test exhaustif, une telle fonction doit être exécutée $2^{(3*32)}$ fois; soit 8×10^{28} fois. Cela représente 2.5×10^{18} années pour un temps d'exécution unitaire de 1 milliseconde. Si, en plus, nous voulons tester cette fonction avec des entrées de types non valides, ce nombre de tests augmente considérablement. On doit donc réduire la taille du JT à un nombre raisonnable de CT.

Par la réduction de la taille du JT, nous rendons le test non seulement faisable mais aussi moins coûteux. D'un autre coté, en limitant le test à un sous-ensemble des DT, nous réduisons l'efficacité du test. Il devient alors essentiel de pouvoir mesurer la capacité d'un JT à dévoiler les fautes dans un PAT. Ainsi, nous pourrons respecter un compromis entre la réduction de la taille du JT et l'aptitude de celui-ci à dévoiler les fautes dans le PAT. Un des moyens pour approcher ce problème est le test par mutation. Le test par mutation est décrit plus en détail au chapitre 2.

Dans le test par mutation on propose une méthode d'évaluation de l'adéquation d'un JT à tester un PAT en particulier. L'idée est d'injecter nous-mêmes des fautes dans des copies du PAT d'une manière représentative de fautes fréquemment présentes dans les logiciels. Chaque copie du programme doit contenir une seule faute injectée. Cette copie est appelée mutant. Un mutant est tué si un CT est capable de le distinguer du PAT. Nous évaluons la qualité du JT en se basant sur le nombre de fautes que ce JT réussit à découvrir.

Même si, l'efficacité du test par mutation est désormais accéptée dans le milieu du test du logiciel, son application reste limitée. Ce manque d'exploitation est dû à des limitations

que le test par mutation connait. En particulier, le test par mutation engendre un grand effort dû à l'exécution des copies erronées du PAT.

Une solution efficace dans le domaine de la réduction des coûts de test est l'automatisation. Les tâches automatisées nécessitent généralement moins de temps et d'effort pour être accomplies. Par conséquent, ces tâches engendrent moins de coûts. De plus, l'automatisation permet d'éviter les fautes humaines dues à l'oubli, aux erreurs de frappe, etc. L'automatisation des activités du test permet d'obtenir des produits logiciels moins chers et de meilleure qualité. Dans les projets de test, l'automatisation devient plus qu'un besoin, elle devient une nécessité.

1.2.2 La Génération Automatique des Données de Test et l'État de l'Art

L'intérêt envers la génération automatique des DT ne cesse de croître au fur et à mesure que des systèmes informatiques de plus en plus puissants font leur apparition. Les travaux qui ont étés fait à ce sujet se basent principalement sur des critères de couverture structurelle. Il s'agit des critères de couverture du test boite blanche. Ces critères sont des mesures d'avancement du test. À titre d'exemple, on peut viser l'exécution de chaque instruction du PAT, ou encore, chaque branche de chaque prédicat dans le graphe de flot de contrôle (control flow graph ou CFG) du PAT.

Les critères de couverture peuvent être utilisés à la fois comme moyen de guider la génération automatique des données tout comme une condition d'arrêt du processus de test.

Miller et Spooner (Miller and Spooner, 1976) furent les premiers à utiliser la recherche locale pour générer des DT pour la couverture de certains chemins dans le PAT. Leurs travaux ont été étendus plus tard par Korel (Korel, 1992) dans sa proposition de méthode dynamique de génération des DT. La méthode commence par générer des DT de manière

aléatoire. Si l'exécution diverge du chemin ciblé, alors on utilise une fonction objectif qui calcule le coût encouru par la non satisfaction de la condition à partir de laquelle on a divergé du chemin cible. Cette mesure est appelée la mesure de la distance des branches. Elle informe sur combien des DT ont été proches d'emprunter la bonne direction en quittant un noeud de contrôle. La recherche vise donc à minimiser ce coût jusqu'à ce que la bonne branche ne soit empruntée. Cette fonction objectif a été améliorée et raffinée pour être utilisée dans plusieurs autres travaux qui utilisent les critères de couverture. Dans Tracey et autres (Tracey et al., 1998b), on remplace la recherche locale par la métaheuristique recuit simulé (RS). Dans ce travail, on utilise une fonction objectif plus sophistiquée qui traite les conditions comprenant des opérateurs relationnels dans des noeuds de contrôle . Dans un travail qui vise la couverture de toutes les branches dans le CFG du PAT, Xanthakis (Xanthakis et al., 1992) a proposé une génération automatique des DT basée sur un algorithme evolutionniste adapté particulièrement au problème du test du logiciel. Le test basé sur de tels algorithmes de recherche est appelé test évolutionniste (TE). De plus en plus de travaux utilisent le TE pour satisfaire différents critères de couverture (Watkins, 1995; Jones et al., 1996; Tracey et al., 1998a; Wegener et al., 2001). L'enquête de Phill Mcminn (McMinn, 2004) est un travail qui récapitule bien l'usage du TE.

La majorité des travaux sur la génération automatique des DT utilisent les informations structurelles du PAT, tel que le CFG, pour guider la recherche des DT. Ces travaux adressaient surtout le test unitaire des fonctions et des méthodes. Un travail récent de Paolo Tonella (Tonella, 2004) a traité de l'utilisation du TE pour tester unitairement les classes des programmes codés dans des langages orientés objet.

En ce qui concerne les tests boite noire, Tracey et autres (Tracey et al., 2000) ont utilisé l'algorithme du RS et les algorithmes génétiques (AG) pour tester la conformité aux spécifications d'un programme écrit en Pascal. Jones (Jones et al., 1995) propose d'utiliser les spécifications comme moyen pour guider la génération automatique des

DT. Il se base, dans ce cas, sur le langage formel de spécifications Z.

À part un travail de Zhan et Clark (Zhan and Clark, 2005) dans lequel ils tentent de générer des DT pour des modèles de Simulink en leurs appliquant des mutations, il ne nous a pas été possible de trouver un travail de génération de DT qui se base sur le test par mutation. Il existe tout de même une proposition de fonction objectif pour le test par mutation développée par L. Bottaci (Bottaci, 2001). Notre fonction objectif s'inspire de ce travail ce qui expliquerait d'éventuelles similarités entre le présent document et le travail de L. Bottaci.

Dans notre approche nous utilisons la métaheuristique colonie de fourmis (CDF). Cette métaheuristique ne semble pas avoir déjà été employée pour la génération des DT. Un travail réalisé par McMinn et Holcombe (McMinn and Holcombe, 2003) utilise la CDF pour retrouver des arbres de chaînage. Dans le domaine du test basé sur l'état, un travail récent de Huaizhong et autres (Li and Lam, 2005) utilise la CDF pour générer des séquences d'entrées pour le test.

Pour profiter des avantages de l'automatisation et d'obtenir en même temps des DT de bonne qualité, un compromis serait de produire en une première phase des DT à l'aide d'outils de génération automatique basés sur des critères de couverture. Par la suite, on procède à une inspection manuelle de ces données en vue de les valider ou de les compléter avec des CT que seule la compréhension de l'intention derrière le code peut inspirer.

Le problème de la définition du comportement référentiel, suppose qu'on dispose d'un moyen pour décrire ce comportement. Or, pour y parvenir, il faudrait que ce comportement soit à la fois prévisible et descriptible ce qui n'est pas toujours le cas.

On doit disposer de moyens, autres que le logiciel lui-même, pour produire une description, que nous souhaitons la plus précise possible, de ce comportement.

Le test par mutation, une technique de test par injection de fautes, propose un moyen de contourner partiellement ce problème pendant la production des DT. En fait, le test par mutation propose de produire des copies du PAT contenant chacune une faute qui y est injectée de manière intentionnelle. Il faut alors exécuter ces copies à l'aide des DT et de comparer le résultat obtenu à ceux fournis par le PAT pour le même JT. Les différences sont soulignées et l'évaluation manuelle de l'ensemble des DT produisant des sorties différentes de celles du PAT permet de savoir si le PAT produit des résultats qui sont bien justes.

1.2.3 Le Problème d'Optimisation et les Métaheuristiques

Dans le problème de génération de DT, non seulement nous essayons de produire des DT de taille réduite, mais aussi nous voulons investir le minimum d'effort possible pour trouver ces DT. Pour le test par mutation, le pire cas est celui d'avoir un CT par mutant. Le cas idéal est de trouver un seul CT qui tue tous les mutants. Dans le cadre de notre travail, nous ne cherchons pas les DT de taille optimale, nous désirons plutôt optimiser l'effort investi dans la recherche de ces DT.

Le coût du test est influencé non seulement par la taille des DT mais aussi par le temps et les ressources engagées dans la recherche de ces DT. Il faut donc réduire l'effort investi pour produire les DT.

Pour aborder les problèmes d'optimisation, les métaheuristiques sont une solution intéressante qui ne cesse d'évoluer. Les métaheuristiques ont fait leurs preuves en matière d'optimisation. Généralement, nous y faisons recours quand il est impossible d'employer des méthodes mathématiques pour la résolution des problèmes d'optimisation. Les métaheuristiques présentent l'avantage d'être flexibles et d'être adaptables à différents problèmes avec une facilité relative.

Il y a une multitude de métaheuristiques ayant chacune des particularités qui les rendent plus ou moins adaptées à la résolution d'un problème particulier.

Un exemple de métaheuristiques largement utilisées sont les AG. Ces derniers s'inspirent de l'adaptation des espèces vivantes à leur environnement à travers une évolution continue d'une génération à une autre. Il existe aussi des métaheuristiques moins connues comme l'escalade de collines à recommencements aléatoires (ECRA). Son principe est de chercher, dans l'entourage d'une position particulière, une meilleure solution que la solution courante. En l'absence d'une meilleure solution locale, on change de manière aléatoire de lieu de recherche en quête d'un endroit permettant plus d'optimisation.

Finalement, il existe des métaheuristiques plus récentes mais non moins intéressantes comme les algorithmes de la CDF. Cette métaheuristique est inspirée du comportement collaboratif des fourmis pour optimiser le chemin qu'elles parcourent en ramenant la nourriture à leur nid.

On pense que cette métaheuristique est sous-employée et que pour certains problèmes elle peut produire des résultats supérieurs à ceux obtenus en utilisant d'autres métaheuristiques. Par exemple, la CDF présente une aptitude naturelle à la parallélisation ce qui n'est pas le cas de bien d'autres métaheuristiques. Pour ces raisons, notre choix c'est fixé sur la CDF comme stratégie d'optimisation. L'emploi de cette métaheuristique à un double objectif :

- trouver le CT capable de tuer un mutant du PAT,
- nous permettre d'optimiser l'effort déployé dans la recherche de ce CT.

1.3 Objectifs et Contributions

L'objectif de ce travail est de réduire le coût du test du logiciel par le moyen de l'automatisation de la génération des DT. Nous proposons un outil de génération de DT de logiciel basé sur le test par mutation et la métaheuristique CDF.

Le recours au test par mutation est pour permettre la production des DT de haute qualité même si celles-ci sont produites de manière automatique. Nous exploitons les avantages du test par injection de fautes pour produire automatiquement des DT de grande qualité. Ce faisant, nous pallions le grand coût de la génération des DT sans détériorer la qualité de ceux-ci.

Nous proposons l'utilisation d'une adaptation de la métaheuristique CDF. Cette adaptation permet d'optimiser l'effort investi dans la recherche des DT produites. Elle pallie ainsi l'un des handicaps du test par mutation : son coût élevé.

Dans notre travail, l'adaptation de la métaheuristique CDF inclut l'usage d'une technique basée sur l'estimation de fonctions de densité de probabilité (FDP). Cette technique permet d'augmenter l'efficacité de l'exploration de l'espace d'entrée à la recherche des DT en dirigeant la recherche vers des zones plus prometteuses.

Nous avons étudié les performances de la solution proposée et nous l'avons comparée à l'usage d'autres métaheuristiques comme l'AG, l'ECRA ainsi qu'à une génération aléatoire des DT.

Pour pouvoir souligner l'avantage de l'utilisation de l'approche basée sur la CDF nous avons donc formulé les hypothèses suivantes :

- Hypothèse nulle, $H0_1$: Il n'existe pas de différence significative dans le nombre de mutants tués en employant notre approche basée sur la CDF et celui des mutants

tués en employant les autres approches alternatives basées sur l'ECRA, l'AG ou la génération aléatoire.

- Hypothèse alternative , $H1$: Notre approche basée sur la CDF tue un nombre de mutants significativement supérieur à celui tué par les approches basées sur l'ECRA, l'AG ou la génération aléatoire.

- Hypothèse nulle, $H0_2$: Il n'existe pas de différence significative entre le coût de l'approche basée sur la CDF et celui des autres approches alternatives basées sur l'ECRA, l'AG ou la génération aléatoire.

- Hypothèse alternative , $H2$: Notre approche basée sur la CDF est plus économique, en matière d'effort investi, que les approches basées sur l'ECRA, l'AG ou la génération aléatoire.

1.4 Organisation du Mémoire

Le présent document est organisé de la manière suivante. Nous commençons par présenter les concepts et les notions nécessaires à la compréhension de la solution proposée (chapitres 2 et 3). Nous continuons avec une formulation mathématique du problème (chapitre 4) et des détails de la solution proposée (chapitre 5) ainsi que de son implémentation (chapitre 6). Nous présentons par la suite les détails de l'expérimentation et les résultats obtenus (chapitre 7). A la fin, nous concluons en récapitulant le contenu essentiel de ce travail et les futures directions de recherches inspirées par ce travail.

CHAPITRE 2

LE TEST PAR MUTATION

2.1 Le Test par Injection de Fautes

Le test par injection de fautes consiste à injecter de manière voulue des fautes dans le PAT. Ceci permet de tester plusieurs aspects du programme. Par exemple, on peut tester la sécurité, la maintenabilité, la réutilisabilité et la tolérance aux fautes.

Le test par injection de fautes sert aussi à évaluer la qualité de DT. Il permet de mesurer la capacité d'un JT particulier à découvrir les fautes présentes dans un PAT.

Le test par injection de fautes inclut plusieurs branches. À titre d'exemple, nous citons la propagation-infection-exécution, l'analyse étendue de propagation, l'analyse adaptive de vulnérabilité et le test par mutation (Voas and McGraw, 1997). Le test par mutation permet d'évaluer la qualité des DT.

Dans le cadre de ce travail, nous nous interessons au test par mutation en tant que moyen de guider la génération de DT.

2.2 Le Test Par Mutation

Le test par mutation est une technique très prometteuse du test de logiciel. Introduite par DeMillo (DeMillo et al., 1978) en 1978, cette technique se veut un moyen efficace d'évaluation de la qualité d'un JT.

Le principe du test par mutation est similaire aux techniques utilisées pour l'estimation

d'une population animale dans une région confinée. Ces techniques consistent à capturer un certain nombre d'animaux présents dans la région en question. Ces animaux sont alors marqués de manière distinctive puis relâchés. Ensuite, la chasse reprend jusqu'à la capture d'un nombre prédéterminé d'animaux. On calcule, alors, la proportion d'animaux marqués parmi ceux capturés. On estime que cette proportion est la même entre tous les animaux initialement marqués et la population totale présente dans la région étudiée. Par exemple, si le nombre d'animaux initialement capturés et marqués est de 300 et que le nombre de ceux marqués parmi 1000 animaux capturés est de 50, alors, on peut estimer la population totale à $\frac{1000}{50} \times 300 = 6000$ animaux. Le nombre d'animaux à marquer et à capturer dépendent d'une estimation initiale approximative de la population et du niveau de confiance que nous voulons avoir dans les résultats.

Dans le test par mutation on mesure la capacité d'un JT à dévoiler les fautes injectées dans le code. Cette mesure est utilisée pour statuer sur la qualité d'un JT. Plus un JT est de meilleure qualité plus il est adéquat pour tester le PAT. Notre critère de qualité est donc un critère d'adéquation du JT.

Le test par mutation a été le sujet d'étude de plusieurs chercheurs. En 1985, Walsh (Walsh, 1985) a trouvé empiriquement que le test par mutation est plus puissant que le test structurel respectant les critères de couverture des branches et des instructions. Frankl et autres (Frankl et al., 1997) et Offutt et autres (Offutt et al., 1996b) affirment que le test par mutation est plus efficace dans la découverte des fautes présentes dans un programme que les méthodes de test basées sur les flux de données. Andrews et autres montrent dans une étude empirique récente (Andrews et al., 2005) que le test par mutation est potentiellement utile pour évaluer et comparer des JT et des critères de test en termes d'efficacité relative par rapport au coût. Ils proposent le test par mutation comme outil pour comparer et évaluer les nouvelles méthodes de test. Ils recommandent également l'adoption du test par mutation dans des situations plus pratiques, par exemple, lorsque nous avons besoin de déterminer empiriquement quels niveaux de cou-

verture sont exigés pour atteindre des taux acceptables de détection.

Dans le test par mutation, nous créons une copie différente, du PAT, pour chaque faute qu'on désire injecter dans le code. Nous obtenons ainsi plusieurs copies du programme contenant chacune une seule faute injectée différente de celles injectées ailleurs dans les autres copies. Chaque copie obtenue s'appelle un mutant. Idéalement nous cherchons à trouver un JT qui soit capable de distinguer le PAT de tous ces mutants. Pour distinguer le PAT d'un mutant, il faut que ce mutant affiche un comportement différent du PAT pour les mêmes DT. Dans un tel cas, on dit que le mutant a été tué par ces DT.

Pour classifier des JT à l'aide de cette technique, nous exécutons le PAT ainsi que tous ces mutants en utilisant ces JT. Le JT qui peut tuer le plus de mutants est considéré meilleur qu'un autre qui en tue moins. Il est possible que deux JT différents tuent le même nombre de mutants. On considère alors que ces deux JT ont la même efficacité même s'il tuent des mutants différents.

Le principe du test par mutation est illustré par l'organigramme de la figure 2.1.

Dans le test par mutation, nous présumons que les DT capables de détecter des fautes simples sont aussi capables de détecter des fautes complexes. Ceci est appelé l'effet de couplage.

2.2.1 La Génération des Mutants et les Opérateurs de Mutation

Le changement apporté au PAT pour obtenir un mutant s'appelle la mutation. La mutation utilise des règles prédéfinies. Ce sont les opérateurs de mutation. Ces opérateurs sont des règles de transformations syntaxiques à apporter au code du PAT pour l'obtention des mutants. L'efficacité du test par mutation réside dans le bon choix des opérateurs de mutation. Ces opérateurs sont inspirés des erreurs communes de programmation pour

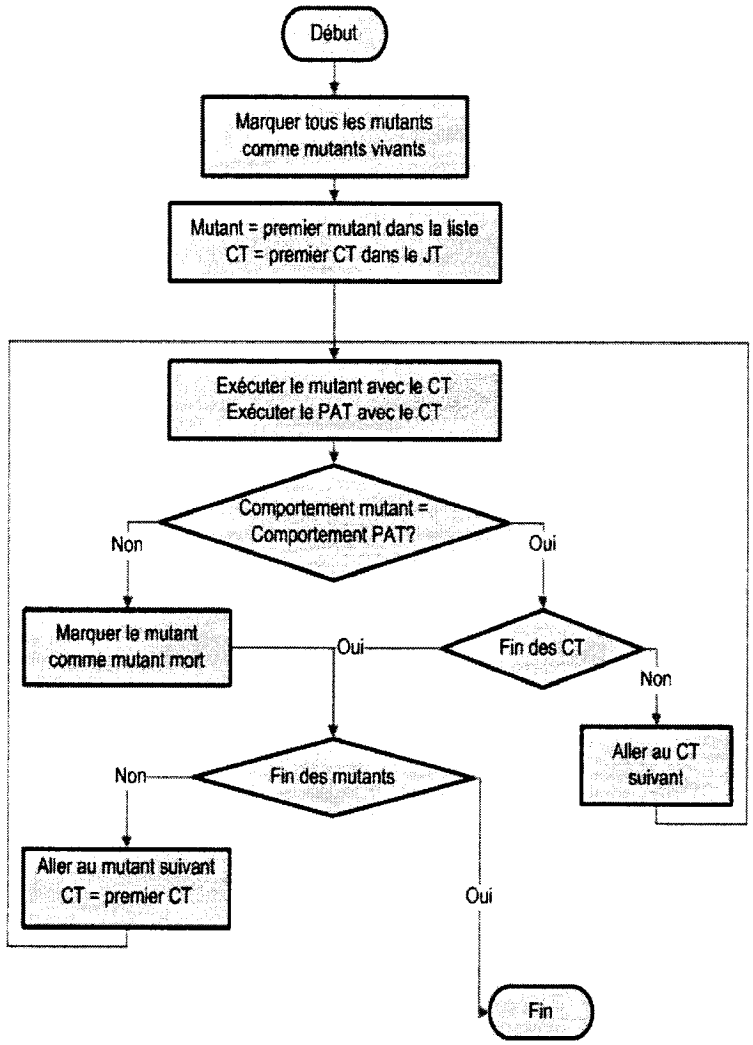


Figure 2.1 Principe du test par mutation

Tableau 2.1 Exemples d'opérateurs de mutation

Operateur	Description
AAR	remplacer une référence à un vecteur par une référence à un vecteur
ACR	remplacer une référence à un veteur par une constante
ASR	remplacer une référence à un veteur par une variable scalaire
CAR	remplacer une constante par une référence à un vecteur
CRP	remplacer une constante par une autre
SDL	effacer une instruction du programme
ROR	remplacer un opérateur relationnel

le langage de développement du PAT. Ils sont choisis de manière à représenter le plus fidèlement possible les erreurs communes commises dans les logiciels. Ils doivent favoriser la sélection de DT de meilleure qualité.

Différents travaux ont étés faits pour proposer des ensembles d'opérateurs de mutation. Parmi ces travaux nous citons celui de J. Offutt et autres. (Offutt et al., 1996a) dans lequel il propose des ensembles réduits d'opérateurs pour la mutation sélective. Dans ce travail, Offutt démontre que la mutation sélective est aussi puissante que la mutation non-sélective. La mutation sélective étant celle basée sur une sélection d'un ensemble réduit d'opérateurs de mutation.

Initialement, les opérateurs de mutation proposés concernaient uniquement des langages procéduraux. D'autres ensembles d'opérateurs ont ensuite été proposés pour les programmes orientés objet et plus récemment pour les programmes concurrents (J.S. Bradbury and Dingel, 2006). Dans le tableau 2.1 nous présentons quelques exemples d'opérateurs de mutation.

Les opérateurs de mutation ont une incidence non seulement sur la qualité du test par mutation mais aussi sur son coût. En fait, le coût du test par mutation dépend du nombre de mutants traités. Or, le nombre de mutants découle du choix des opérateurs de mutation et du nombre de points du programme où ces opérateurs peuvent être appliqués.

Le processus d'injection de fautes dans le programme peut être manuel ou automatisé. Dans le cas où la génération est faite de manière automatisée, nous ne pouvons pas garantir qu'il existe des données d'entrée pour lesquelles tous les mutants obtenus se comporteront différemment du programme original. Les mutants qui ont le même comportement que le programme original pour tout l'espace d'entrée sont appelés des mutants équivalents ou secondaires. Ces mutants sont donc différents syntaxiquement du programme original mais restent fonctionnellement équivalents à celui-ci. L'identification de tels mutants nécessite le parcours de tout l'espace de recherche. C'est donc, à son tour, un problème complexe. Ce problème est source d'intérêt de beaucoup de chercheurs dans le domaine du test par mutation.

Le choix des fautes injectées dans chaque mutant influence directement la qualité du test par mutation. Les opérateurs de mutation essayent de produire des fautes semblables à ceux introduits dans le programme durant son développement. Plus ces fautes injectées sont représentatives des fautes réelles plus les DT qui y sont sensibles sont de meilleure qualité.

Un générateur de mutant peut être vu comme un parseur qui traverse la structure du programme en appliquant des règles d'injection des fautes aux nœuds appropriés. On parcourt les nœuds à la recherche de nœuds auquel une des règles de mutation s'applique. Chaque fois qu'un tel nœud est rencontré :

1. on apporte la modification correspondant à la règle de mutation,
2. on sauvegarde cette copie mutée du programme,
3. on rétablit l'état original du code et
4. on poursuit la recherche de nœuds mutables.

Il existe plusieurs outils de génération de mutants pour différents langages de program-

mation. Sur Internet, nous pouvons aussi trouver des dépôts de programmes à tester incluant leurs mutants.

Dans notre cas, nous avons utilisé un outil de génération de mutants pour les programmes écrits en java : MuJava. Développé par Offutt et Kwon (Offutt et al., 2004), cet outil permet non seulement de générer les mutants mais aussi de les exécuter à l'aide de DT fournies par l'utilisateur. Pour notre travail nous avons utilisé MuJava uniquement pour obtenir les mutants de notre PAT. Cet outil est diponible à l'adresse : http://ise.gmu.edu/ofut/mujava/

2.2.2 Le Score de Mutation

Dans le test par mutation, le meilleur JT est celui qui tue le plus de mutants du PAT. D'où l'utilité de la mesure du nombre de mutants tués par rapport au nombre total de mutants non équivalents. Cette mesure, dite le score de mutation, sert à classifier les JT. Le score de mutation est donc une mesure de l'adéquation d'un JT. On le calcule par l'équation 2.1.

$$ScoreDeMutation = \frac{NombreDeMutantsMorts}{NombreTotaleDeMutantsNonEquivalentsDuPAT} \quad (2.1)$$

Comme on le verra dans la section 4.2, quoi que cette mesure soit intéressante pour classifier les DT, elle n'est pas suffisante pour guider la recherche et la génération des DT.

2.2.3 Le Problèmes et les Défis du Test par Mutation

Le test par mutation fait face à un certain nombre de problèmes. Entre autre, l'exécution des mutants qui engendre des coûts importants. Contrairement au test de couverture où la seule exécution du PAT suffit, dans le test par mutation, pour chaque CT on doit exécuter le PAT et tous ces mutants non encore tués. Ceci peut s'avérer un effort important. Pour pallier ce problème, il existe trois écoles de pensée ou axes de recherche : 'en faire moins' (do few), 'exécution intelligente' (do smarter) et 'exécution rapide' (do faster) (May et al., 2003). Les premiers cherchent à exécuter moins de mutants en essayant de préserver la qualité du test par mutation. Cette catégorie inclut la mutation sélective (Mresa and Bottaci, 1999) et l'échantillonnage des mutants (Budd, 1980; May et al., 2003). La mutation sélective vise à utiliser un ensemble réduit d'opérateurs de mutation. Si on arrive à utiliser moins d'opérateurs de mutation, on arrive à réduire le nombre de mutants et par suite on réduit l'effort d'exécution. L'échantillonnage des mutants réduit aussi le nombre de mutants à exécuter. On choisit un sous-ensemble de mutants à exécuter parmi ceux générés sans détériorer la qualité du test par mutation.

L'exécution intelligente cherche à paralléliser l'effort d'exécution sur plusieurs systèmes ou encore à ne pas exécuter la totalité du code du mutant. Dans ce cadre, nous citons la mutation faible (Howden, 1982). Dans la mutation faible, on considère suffisant d'obtenir un état du mutant qui soit différent de celui du PAT après l'exécution de l'instruction mutée.

Dans la famille des 'exécution rapide' , nous essayons de générer et d'exécuter les mutants de la manière la plus rapide possible. On y compte la mutation basée sur les schémas (Offutt and Untch, 2000) et la compilation séparée (Untch et al., 1993).

Un autre problème du test par mutation est celui d'identifier et d'éliminer les mutants équivalents. Pour illustrer ce problème, considérons la fonction de la figure 2.2.

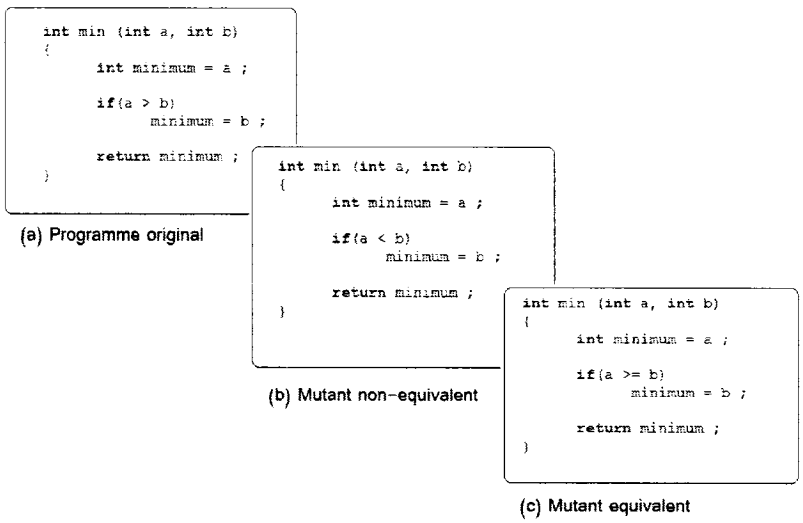


Figure 2.2 Exemple de mutants équivalents et non-équivalents d'un programme

La valeur de retour du premier mutant sera différente de celle du programme original pour toute valeur de x autre que la valeur zéro. Donc tout choix de x autre que zéro nous permet de distinguer le mutant du PAT. Ceci n'est pas le cas du deuxième mutant. En fait, pour le deuxième mutant, il n'existe pas de valeur de x pour laquelle la sortie va être différente de celle du PAT. Donc, si nous nous basons sur la valeur retournée pour distinguer les mutants, ce mutant est donc impossible à démasquer. C'est donc un mutant équivalent. Identifier de manière automatique les mutants équivalents (Adamopoulos et al., 2004) est un autre moyen pour réduire le coût du test par mutation.

Dans notre travail, pour identifier les mutants équivalents, nous avons eu recours à un compromis entre l'exécution automatique des mutants et l'inspection manuelle du code. Après plusieurs exécutions, nous avons pu identifier un sous-ensemble de mutants qui résistait à toute tentative de démasquage. Ce sous-ensemble étant inférieur au nombre total de mutants obtenus par la génération automatique des mutants, son inspection

manuelle n'a pas été une tâche ardue. Il était question de lire le code autour de la phrase mutée et d'essayer de trouver une combinaison qui puisse passer par la phrase mutée et produire un résultat de calcul différent de celui obtenu au même point du PAT et qui se propage jusqu'à la sortie du mutant.

Le premier travail à avoir proposé le test par mutation comme base à la génération des DT est la thèse de doctorat de J. Offut (Offutt, 1988). La technique est appelée Technique de Génération de DT basée sur les Contraintes (Constraint-Based Test data generation technique ou CBT). La CBT est basée sur l'observation que, pour qu'un CT puisse tuer un mutant il faut qu'il satisfasse les trois conditions suivantes :

1. l'exécution doit passer par l'instruction mutée,
2. l'exécution de l'instruction mutée dans le mutant doit produire un résultat de calcul différent de celui de la même instruction dans le PAT
3. la différence de calcul de la condition précédente doit être visible à la sortie du mutant et du PAT.

La CBT souffre de plusieurs faiblesses dues, entre autre, à la procédure de recherche de DT trop simple. Pour pallier les faiblesses de la CBT, une autre technique appelée la réduction dynamique de domaine (Dynamic domain reduction ou DDR) a été proposée par le même auteur (Offutt et al., 1999). La DDR reprend la CBT en proposant une meilleure procédure de recherche qui permet le partage du domaine de recherche par bissection.

Le test par mutation est une stratégie de test originale qui présente de grands avantages par rapport à d'autres stratégies de test boite blanche. Elle est tout de même très lourde et nécessite plus d'effort que d'autres stratégies. Malgré ce problème, la mutation reste une stratégie de grand intérêt grâce à la qualité élevée des DT qu'elle produit et de sa

capacité particulière à évaluer l'adéquation des DT.

CHAPITRE 3

LES MÉTAHEURISTIQUES UTILISÉES

Le problème de génération de DT est un problème d'optimisation. On cherche à obtenir les DT de la taille la plus réduite avec le moins d'effort possible, sans affaiblir l'efficacité du test. Pour ce type de problème, l'emploi des métaheuristiques s'avère une solution appropriée.

Les métaheuristiques sont un ensemble d'algorithmes regroupés en familles qui servent à résoudre des problèmes d'optimisation pour lesquels on ne connait pas de méthode de résolution classique qui donne de meilleurs résultats. Il y a plusieurs métaheuristiques. Parmi les métaheuristiques les plus utilisées on trouve les algorithmes génétiques, le recuit simulé, la recherche taboue, la colonie de fourmis et l'escalade des collines. Pour notre travail, et pour pouvoir évaluer les performances de notre approche nous avons utilisé les métaheuristiques décrites dans les paragraphes suivants.

3.1 L'Algorithmes d'Escalade des Collines à Recommencement Aléatoire

L'algorithme d'escalade des collines est une métaheuristique simple basée sur le recherche locale. Dans cette métaheuristique, nous tentons d'améliorer une solution initiale en cherchant une meilleure solution dans son voisinage. Si une meilleure solution est trouvée, celle-ci devient la solution adoptée ou la solution courante. Ce processus continue de manière itérative jusqu'à l'atteinte d'un critère d'arrêt. Si la solution courante est la meilleure parmi toutes ses voisines, on peut choisir de diversifier la recherche en la continuant à un autre endroit choisi aléatoirement. Cette variante s'appelle l'ECRA. C'est à cette variante de la métaheuristique escalade des collines qu'on a choisi de se

comparer. Dans l'ECRA, il faut définir la notion de voisinage entre solutions. En général, une solution voisine correspond à une modification partielle de la solution courante.

3.1.1 Adaptation de l'Escalade des Collines à Recommencement Aléatoire au Test par Mutation

Pour adapter l'ECRA à un problème en particulier, il faut définir :

- la notion de voisinage
- la fonction objectif
- le critère d'arrêt

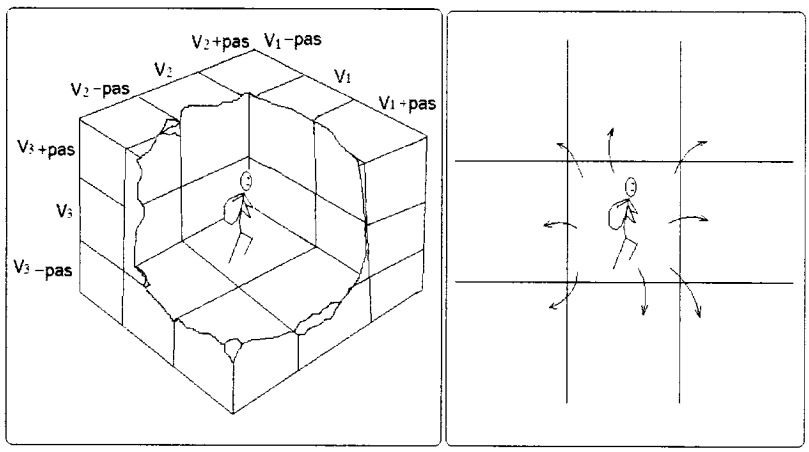
L'objectif étant de trouver un CT qui tue un mutant particulier, notre solution initiale consiste donc en un CT choisi aléatoirement. Pour définir le voisinage, nous avons besoin de spécifier ce qu'est le voisin d'un CT. Un CT est formé principalement d'un ensemble de valeurs attribuées aux paramètres du PAT. On définit alors le voisin d'un CT un autre CT différent du premier tel que les valeurs du deuxième cas de test sont égales ou voisines de celles du premier cas de test. Deux valeurs voisines sont deux valeurs séparées par un intervalle jouant le rôle d'un pas. À titre d'exemple, pour le programme Triangle qui nécessite trois paramètres, deux CT voisins peuvent être $CT1$ composé des valeurs $v1$, $v2$ et $v3$ et $CT2$ composé des valeurs $v1$, $v2$+pas , $v3$-pas. On rappelle qu'un CT est composé non seulement des valeurs des paramètres mais aussi des valeurs attendues en retour. Ces valeurs de retour dépendent du problème résolu par le PAT et ne sont donc pas concernés par cette notion de voisinage.

En adoptant cette notion de pas, chaque valeur possède donc 2 voisins. Le nombre total de possibilités est égale au cube du nombre de paramètres du PAT. Un CT voisin doit

être différent du CT lui même, le nombre de voisins possible d'un CT est donc donné par 3.1.

$$NombreDeVoisins = (NombreDeParameres)^3 - 1 \tag{3.1}$$

Toujours pour le cas du programme Triangle, le voisinage d'un CT serait composé de 26 CT qui peuvent êtres représentés par un cube tel que dans la figure 3.1. Dans la même figure, nous voyons le cas d'un voisinage bi-dimentionnel.



(a) cas du programme Triangle (b) cas de deux paramètres

Figure 3.1 Voisinage d'un CT pour la méhtaheuristique ECRA

Afin de tuer un mutant donné, l'ECRA commence par choisir un CT aléatoire comme première solution. La qualité du CT est évaluée par la même fonction objectif utilisée dans la CDF. Des détails de cette fonction sont donnés dans la section 555. L'ECRA essaie d'améliorer le CT courant en se déplaçant vers des meilleurs points dans un

voisinage de la solution courante. Ce processus itératif continue jusqu'à ce que le mutant soit tué ou qu'on ait atteint un nombre maximal d'évaluations de solutions. Il est aussi possible que l'algorithme ne trouve plus de meilleure solution dans un certain voisinage. Dans ce cas si le mutant est encore vivant et qu'on n'a pas dépassé le nombre maximal d'évaluations de solutions, on reprend la recherche à un autre endroit choisi aléatoirement. Ce changement de lieux de recherche s'appelle un saut. Pour l'ECRA il serait intéressant d'évaluer le nombre de sauts par rapport au nombre totale d'évaluations. Ce rapport indique combien notre recherche est proche d'une recherche aléatoire. Donc, plus ce rapport est petit plus l'efficacité de l'ECRA est grande.

3.2 Les Algorithmes Génétiques

Les algorithmes génétiques sont des algorithmes évolutionnistes. Proposés initialement par J. H. Holland (Holland, 1992), les AG sont parmi les métaheuristiques les plus utilisées. Ils sont inspirés de la biologie évolutionniste. Ils se basent sur les techniques de recombinaison. Les AG sont largement utilisés dans la résolution de beaucoup de problèmes de recherche notamment dans le domaine du génie logiciel.

Dans les AG, nous commençons par créer une population de solutions. Cette population initiale peut être choisie de manière aléatoire ou non. Ensuite, nous faisons évoluer cette population en utilisant les techniques de recombinaison. Nous espérons que cette évolution conduise à l'apparition de plus en plus de solutions de meilleure qualité dans les nouvelles générations. Cette évolution continue donc jusqu'à ce qu'on produise une solution satisfaisante selon le temps ou l'effort qu'on est prêt à investir dans cette recherche. Dans les AG, une solution est représentée de manière à permettre sa manipulation dans des opérations de recombinaison. Il faut donc coder les solutions sous forme de chromosomes composés de gènes. Les opérations appliquées sur ces chromosomes sont la sélection, la mutation et le croisement. Ces opérations sont aussi appelées

opérateurs. Ces opérateurs prennent en entrée un ou deux chromosomes et produisent autant de chromosomes que ceux reçus en entrée. L'application de l'une ou l'autre des opérations de recombinaison dépend d'une probabilité fixe. Cette probabilité permet de contrôler la vitesse de convergence et la diversification dans la recherche de solution. En général, nous appliquons le croisement sur une majorité de la population alors que la mutation n'est appliquée que sur une minorité de celle-ci.

Le principe des AG correspondent à l'algorithme suivant :

1. Création de la population initiale
2. Évaluation de l'ensemble des solutions
3. Sélection
4. Croisement
5. Mutation
6. A-t-on atteint le critère d'arrêt? Si oui fin
7. Constituer la nouvelle génération et continuer en 2

3.2.1 Les Opérateurs Génétiques

3.2.1.1 La sélection

La sélection est l'opération par laquelle nous décidons quels chromosomes vont servir à produire les nouveaux chromosomes qui feront partie de la prochaine génération de solutions. Elle peut être basée sur un choix objectif ou subjectif. Le deuxième cas nécéssite généralement moins d'effort. Dans la plupart des cas, le choix des chromosomes est basé sur la fonction objectif de la métaheuristique. La sélection peut se faire

de plusieurs manières selon notre désir de donner plus ou moins de chances aux chromosomes faible. Dans notre cas, elle se fait selon un algorithme de roulette de casino qui donne à chaque chromosome des chances de sélection proportionnelles à sa qualité fournie par la fonction objectif.

3.2.1.2 Le Croisement

Le croisement est l'opération la plus importante des AG. Dans le croisement, deux individus sont choisis pour se reproduire et produire deux nouveaux chromosomes. Les chromosomes produits sont composés à partir de trançons de leurs parents. Couramment nous décidons d'une position à partir de laquelle chaque chromosome est copié dans un seul des deux enfants. Le reste des chromosomes est ensuite copié dans le second enfant. À la fin, on obtient des enfants ayant chacun une partie de chaque parent. Il est possible de choisir plus qu'un seul endroit de croisement. Nous utilisons souvent le croisement à un point (pour un seul endroit de croisement) ou à deux points (pour deux endroits de croisement). Ces deux croisements sont illustrés dans la figure 3.2.

Il existe aussi des variantes de l'opération de croisement dans lesquelles nous attribuons un point de croisement différent pour chaque parent. Dans ce cas, les enfants peuvent être composés d'un nombre de gènes différent de ceux de leurs parents.

On distingue aussi le croisement uniforme et le croisement non-uniforme. Dans le croisement uniforme, pour chaque gène de l'enfant, le parent source est choisi selon une probabilité fixe habituellement égale à 0.5. Dans le croisement non-uniforme, cette probabilité est variable.

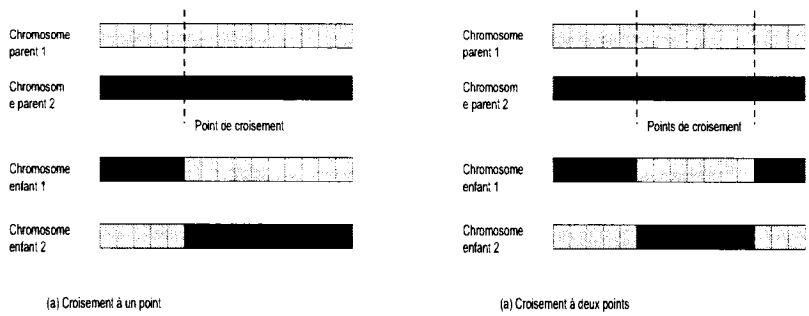


Figure 3.2 Opération de croisement dans les AG

3.2.1.3 La Mutation

La mutation est une opération analogue à celle existante dans la biologie. Elle consiste à remplacer certains gènes d'un chromosome choisi pour produire un nouveau chromosome. Contrairement au croisement, la mutation permet de créer des solutions qui ne sont pas forcément composées de fragments d'autres solutions actuellement présentes dans la population. La mutation est donc l'opération qui permet la diversification dans la recherche des solutions. L'intérêt de la diversification et celui d'éviter d'être pris dans le piège d'un minimum (ou maximum) local. Ceci dit, la mutation n'est appliquée qu'à une petite proportion de la population.

Dans le cas de gènes de type binaires, nous décidons de manière aléatoire si chaque gène doit être inversé ou non. En général, le gène est remplacé par une autre valeur parmi les valeurs possibles qu'il peut avoir. Le remplacement des gènes du chromosome se fait couramment de manière aléatoire. Il se peut tout de même que ce remplacement se fasse selon une règle, qu'on espère, conduirait à une meilleure diversification ou à des solutions de meilleure qualité.

3.2.2 Adaptation de l'AG au Test par Mutation

Pour appliquer et adapter les AG à notre travail, nous avons besoin de définir ce qu'est un chromosome et les gènes qui le constituent. Un chromosome étant une solution du problème, donc pour nous, un chromosome est un ensemble de valeurs d'entrée du PAT. C'est un CT sans la valeur de retour. Dans ce cas, un gène est un paramètre du PAT.

Pour générer la population initiale, nous commençons par générer un individu choisi aléatoirement à partir du domaine D du PAT. Chaque individu représente un chromosome. Les gènes sont des valeurs des paramètres d'entrée du PAT. Dans un processus itératif, GA essaye d'améliorer la population d'une génération à l'autre. Des individus d'une génération sont sélectionnés selon leur qualité retournée par la fonction objectif. Sur ces individus, nous appliquons les opérations de recombinaison c.-à-d. le croisement et la mutation. Une nouvelle génération est ainsi produite constituée par les l meilleurs individus de la génération précédente et de la progéniture obtenues à partir des opérations de recombinaison. La nouvelle génération comprend alors les n meilleurs individus générés. Le processus itératif continue jusqu'à ce que le critère de d'arrêt soit rencontré. Dans notre cas, le critère d'arrêt est que le mutant soit tué ou que le nombre maximal d'évaluations soit atteint. Dans notre expérience, nous avons choisi un croisement uniforme : dans l'individu généré, la valeur du i^{eme} gène sera la valeur du i^{eme} gène de l'un des deux individus parents choisi aléatoirement. La mutation a été exécutée par une modification aléatoire d'un gène de l'individu. Une deuxième alternative de l'opérateur de mutation serait de remplacer l'individu par un de ces voisins tels que définis dans l'ECRA.

3.3 La Métaheurisitique Colonie de Fourmis

3.3.1 L'intelligence collective

L'intelligence collective est l'aptitude d'un groupe de membres à résoudre un problème sur lequel ils ne possèdent individuellement que des vues partielles. La résolution se fait à travers des mécanismes de communication et collaboration. Les membres adoptent chacun un comportement qui ne résout pas le problème en soi mais qui fait en sorte que l'ensemble des actions des membres du groupe y parviennent. Ainsi, à partir des comportements simples et primitifs des membres, nous arrivons à résoudre des problèmes de grande complexité.

L'intelligence collective est présente, entre autre, chez les insectes qui vivent en collectivité. Elle est liée à la présence d'un ensemble de conditions : - les membres du groupe ne détiennent qu'une partie de l'information totale - le comportement des membres est régie par un ensemble de règles simples que tous les membres respectent avec fidélité - chaque membre collabore avec un ou plusieurs autres membres du groupe - la résolution du problème est un objectif global du groupe.

L'intelligence artificielle est observée chez les insectes vivant en collectivité comme les abeilles et les fourmis. Dans ce cas, elle est appelée intelligence des insectes (swarm intelligence). Elle est aussi observée chez des animaux mammifères comme les otaries qui chassent collectivement les phoques.

Notre travail utilise l'intelligence des insectes. Plus précisément, nous nous interessons à une métaheuristique qui s'inspire de l'intelligence des fourmis.

3.3.1.1 La Colonie de Fourmis

La métaheuristique colonie de fourmis fût introduite par Marco Dorigo dans sa thèse de doctorat (Dorigo, 1992). Cette métaheuristique imite un cas concret d'intelligence collective. Inspirée du comportement des fourmis, elle se penche sur leur collaboration pour identifier le plus court chemin entre une destination (la nourriture) et leur habitation. Il a été observé que les fourmis arrivent à trouver le chemin le plus court, en termes de longueur et de contournements d'obstacles, entre leur habitation et la localisation de la nourriture rapportée par l'un des membres de la colonie. L'étude de ce comportement a donné lieu à une nouvelle métaheuristique. Cette métaheuristique est de plus en plus utilisée car elle s'est avérée efficace pour la résolution de plusieurs problèmes d'optimisation. Aujourd'hui on compte plusieurs algorithmes inspirés de cette métaheuristique.

La collaboration des fourmis commence lorsque l'une d'elles identifie la localisation de la nourriture. Cette fourmi, en revenant au nid, dépose sur le sol une substance volatile dite la phéromone. Les autres fourmis, sachant la direction vers laquelle il faut aller, tentent de rejoindre la localisation de cette nourriture en essayant de suivre le chemin marqué par la phéromone. À la rencontre d'obstacles sur leurs chemins, les fourmis doivent prendre des décisions pour les contourner tout en essayant de passer par le chemin le plus court vers la nourriture. Les fourmis finissent par suivre différents chemins. Au retour vers le nid, chaque fourmi propose son propre chemin comme étant le chemin à emprunter en y déposant encore de la phéromone. Après un certain moment, on observe une convergence des chemins empruntés par les fourmis vers un seul chemin : celui le plus court. Ceci est dû au fait que la phéromone, qui s'évapore à la même vitesse, disparait plus vite des chemins plus longs. En même temps, le chemin le plus court voit passer plus de fourmis dans le même intervalle de temps qu'un autre qui est plus long. Ce chemin garde donc une trace plus nette et visible. Progressivement, ce

court chemin va être encore plus adopté que les autres alternatives moins intéressantes. Finalement le choix de la majorité va être adopté.

Ce comportement schématisé dans la figure 3.3 est résumé par les étapes suivantes :

1. la fourmi qui trouve la nourriture revient au nid en déposant de la phéromone sur son chemin
2. les fourmis tentent de retrouver cette nourriture en empruntant quasiment le même chemin en évitant de manière différente les obstacles
3. après avoir atteint la nourriture, les fourmis reviennent au nid en déployant de la même manière la phéromone sur leurs chemins de retour
4. les fourmis repartent à la recherche de la nourriture en empruntant le chemin ayant la teneur de phéromone la plus importante
5. à cause de l'évaporation de la phéromone, les chemins convergent vers le chemin le plus emprunté
6. la phéromone disparait progressivement de tout chemin alternatif au chemin le plus court

L'adoption de ce comportement dans les problèmes de recherche essaye d'imiter le plus fidèlement possible le comportement des fourmis. L'espace de recherche est partagé en zones voisines entre lesquelles il existe des chemins sur lesquels la phéromone sera déposée. Les fourmis virtuelles sont appelées à parcourir chacune un chemin qu'elle juge le plus court vers l'objectif. À chaque étape, les fourmis se déplacent entre leur emplacement courant et un emplacement voisin. Elles sont appelées à évaluer la meilleure destination partielle pour que le chemin total soit optimal. N'ayant pas de vue globale sur l'espace à parcourir, les fourmis virtuelles décident de leur destination future en

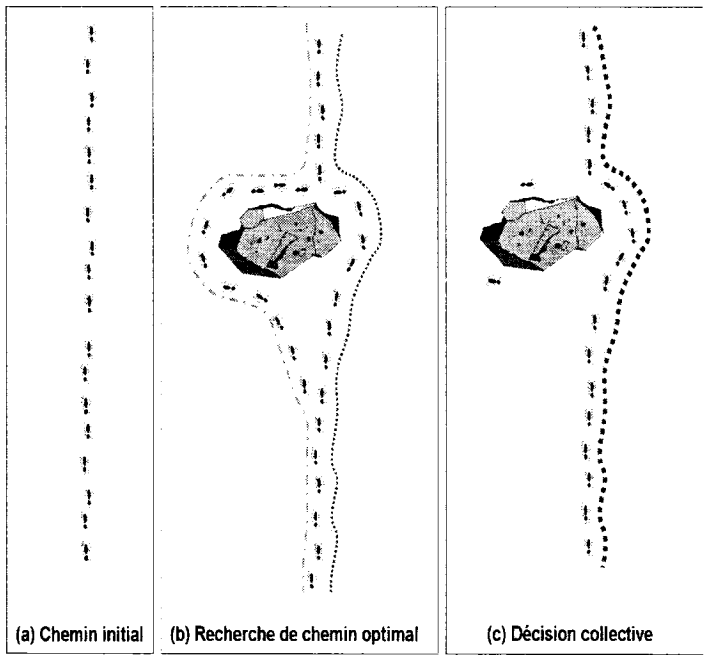


Figure 3.3 Comportement collaboratif des fourmis

fonction de deux critères : - la quantité de phéromone virtuelle présente sur chaque bout de chemin - l'adéquation du bout de chemin choisi par rapport à l'objectif global selon leurs point de vue.

Une fois la destination finale atteinte, les fourmis déposent la phéromone sur le chemin qu'elles viennent de parcourir. La quantité de phéromone déposée est proportionnelle à la qualité du chemin emprunté. Cette qualité doit être évaluée grâce à une fonction prédéfinie dite la fonction d'adéquation ou fonction objectif.

À la fin de chaque cycle constitué de la construction d'une solution par chaque fourmi, on simule l'évaporation naturelle de la phéromone en réduisant sa quantité partout où elle se trouve. Une réduction de la phéromone au cours du temps (l'écoulement du temps ou le nombre de sauts des fourmis) complète cette simulation de la réalité et fait en sorte que le chemin le plus parcouru est celui qui garde le plus de phéromone.

À la fin de chaque cycle, on procède alors à une mise à jour de la phéromone. Cette mise

à jour inclut l'ajout et l'évaporation de la phéromone. Pour chaque chemin reliant deux zones voisines, on lui ajoute autant de phéromone que les fourmis voudraient y déposer dépendament de si, il fait partie ou non de leur chemin choisi. Au même chemin on retire autant de phéromone qu'il faut pour simuler le phénomène d'évaporation naturelle de la phéromone. La quantité de phéromone sur un chemin, à la fin d'un cycle $t+1$, est égale à la quatité qui s'y trouvait au cycle t réduite de ρ % à laquelle on rajoute toute la phéromone que les fourmis doivent y déposer pour le cycle $t+1$. Ce calcul est fourni par la formule suivante :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{3.2}$$

où m est le nombre de fourmis, i et j sont les indexes de deux zones voisines.

ρ est un paramètre de l'algorithme. Sa valeur décide de la vitesse dans le temps avec laquelle disparait l'avantage acquis par une solution précédement choisie. Pour une résolution plus efficace ou plus rapide, on peut agir sur le critère de décision des fourmis dans leurs choix du chemin à adopter pour rejoindre la prochaine zone. Dans ce critère on inclut une heuristique propre au probléme à résoudre. Pour favoriser une diversification de la recherche, cette heuristique peut, par exemple, donner plus de chance aux destinations les moins parcourues. Pour accélérer la recherche, l'heuristique peut avantager des destinations qu'on pense ont plus de chance d'aboutir à de meilleures solutions selon la nature de notre problème.

La fonction qui calcule la probabilité de choix de la prochaine destination de la fourmi s'appelle la règle de déplacement. Elle combine donc la phéromone et l'heuristique. Cette probabilité est calculée par 3.3.

$$p_{ij}^{k}(\mathrm{t}) = \begin{cases} \frac{\tau_{ij}(t)^{\alpha}.\eta_{ij}^{\beta}}{\sum_{l \in J_i^k} \tau_{ij}(t)^{\alpha}.\eta_{ij}^{\beta}} & \mathrm{sij} \in \mathrm{J_i^k} \\ 0 & \mathrm{sij} \notin \mathrm{J_i^k} \end{cases} \tag{3.3}$$

Où : J_i^k est l'ensemble de déplacements possibles pour la fourmi k qui se trouve à la position i et η_{ij} est la valeur de l'heuristique pour chaque déplacement de la position i à la position j. α et β sont les paramètres qui permettent de contrôler l'influence relative de la phéromone et de l'heuristique dans la prise de décision de déplacement.

À chaque itération de l'algorithme de la CDF, chaque fourmi propose sa propre solution au problème. On procède alors à la mise à jour de la phéromone. On évalue donc la qualité de chaque solution retournée et on ajoute de la phéromone sur les chemins que représentent les solutions proposées.

3.3.2 Adaptation de la CDF au Test par Mutation

Adapter la métaheuristique CDF à un problème quelconque revient à définir une structure de graphe dans le domaine des solutions, une stratégie de mise à jour de la phéromone et une fonction de transition qui gére le déplacement des fourmis. Le graphe à définir est celui qui sera parcouru par les fourmis pendant leur construction d'une solution. Dans notre cas, une solution est un CT constitué des valeurs des paramètres d'entrée du PAT. Pour le problème de génération de DT pour le test par mutation on peut, alors, faire les choix suivants :

1. Les paramètres du PAT constituent des destinations fixes par lesquelles chaque fourmi doit passer
2. Les valeurs pouvant être attribuées au prochain paramètre à visiter représentent les différents chemins que la fourmi peut entreprendre pour visiter le dit paramètre

La fourmi attribue une valeur au premier paramètre du PAT. Ce choix constitue le chemin qui la mène vers le paramètre suivant. Elles doit alors attribuer une valeur à celui-ci. Ce processus continue jusqu'à l'attribution d'une valeur au dernier paramètre. Cette attribution complète la construction de la solution proposée par cette fourmi et conduit la fourmi de nouveau vers le premier paramètre du PAT. La fourmi est alors prête pour une prochaine itération. La qualité de la solution retournée par la fourmi est évaluée et la phéromone est déposée sur les valeurs constituant cette solution.

Pour contrôler la diversification de l'espace de recherche, nous commençons par des itérations aléatoires. Pendant ces étapes aléatoires, les décisions que prennent les fourmis pour emprunter un chemin ou un autre sont complètement arbitraires. Ceci dit, les fourmis déposent la phéromone pour qu'elle serve pendant les prochaines étapes de recherche comme élément de prise de décision. Le comportement similaire chez les fourmis réelles est celui des fourmis éclaireuses. Ce sont les fourmis en charge de trouver les aliments que les autres fourmis doivent rapporter au nid. Ces étapes permettent une exploration de l'espaces de recherche sans se limiter à la recherche dans un ensemble favorisé dès le début.

Une fois les informations sur la qualité des chemins sont distribuées de manière suffisante, on commence à prendre en considération la phéromone déposée pour décider du chemin à emprunter.

La CDF classique n'est pas appropriée pour une recherche dans un espace continu. Pour pouvoir construire une solution dans un espace de recherche continu, la CDF nécessite un mécanisme de construction des solutions différent de celui qui se base sur un graphe statique. Notre solution à ce problème est exposée dans le chapitre 4

CHAPITRE 4

MODÉLISATION MATHÉMATIQUE DU PROBLÈME

4.1 La Génération Automatique des Données de Test

Cette section introduit la formulation mathématique du problème de génération automatique des DT, que nous souhaitons capable de tuer l'ensemble des mutants d'un PAT. Ce problème revient à un problème d'optimisation qui repose sur une modélisation structurelle du PAT.

Soit P_g un PAT et $I = (x_1, x_2, ..., x_k)$ le vecteur de ces paramètres d'entrée. Chaque paramètre x_i prend sa valeur dans un domaine $Di, i = 1, 2, ..., k$. Le domaine du programme est donc $D = D_1 \times D_2 \times ... \times D_k$.

D'un autre coté appelons R l'ensemble des opérateurs de mutation appliqués au PAT P_g. Chaque opérateur r dans R représente un type différent de fautes typiques de programmation dans le langage de programmation de P_g. L'application de r au PAT produit autant de nouveaux mutants que le nombre d'emplacements de P_g la où r est applicable.

On suppose que l'application de tous les opérateurs appartenant à R au programme P_g produit N mutants M_1,M_2,...,M_N. On écrit $M_j = r_l(P_g)$ pour décrire le mutant M_j obtenu à la l^{ieme} application de l'opérateur de mutation $r \in R$ au programme P_g. L'instruction de M_j qui comprend la différence entre M_j et P_g est dite l'instruction mutée. Elle sera notée s_m. Dans ce document, on emploira cette appelation pour désigner à la fois l'instruction avant mutation dans le PAT et l'instruction obtenue après mutation dans le mutant. On distinguera la ligne de code visée selon le contexte.

Le problème de la génération de DT dans le contexte du test par mutation est donc la recherche de l'ensemble de DT qui maximise le nombre de mutants tués du PAT. En d'autres termes, il faut trouver les valeurs à attribuer aux paramètres x_i du PAT P_g de façon à ce qu'un nombre maximale parmi les mutants soit tué. Les valeurs d'entrées recherchées formeront des CT et l'ensemble de ces CT formera le JT qui servira à tester le PAT.

4.2 La Fonction Objectif pour le Test par Mutation

Un problème d'optimisation est un problème mathématique dans lequel on cherche à minimiser ou maximiser une fonction réelle. L'optimisation se fait en choisissant, parmi les valeurs possibles, des valeurs pour les variables de cette fonction. Le problème de minimisation, se formule mathématiquement de la manière suivante :

Étant donné une fonction : $f : A \rightarrow R$ rechercher un élément : $x_0 \in A$ telque $f(x_0) \leq f(x)$; $\forall x \in A$

La fonction f est dite la fonction objectif, la fonction du coût ou la fonction de fitness. Cette fonction prend en argument la solution proposée et fournit en retour une valeur réélle qui informe sur le coût ou la qualité de cette solution. Dans un problème de minimisation, la solution qui regénère la plus petite valeur de retour de la fonction objectif est la solution recherchée.

Le score de mutation est un bon moyen pour quantifier l'adéquation d'un JT pour tester le PAT, mais, considérant un seul mutant à la fois, cette métrique est booléenne. Il nous dit si le mutant est tué ou s'il est toujours vivant. Le score de mutation est incapable de nous fournir plus de détails sur combien un CT en particulier est proche de tuer un mutant. Ceci le rend inapproprié comme fonction objectif pour notre recherche.

Dans notre problème, les solutions recherchées sont des solutions capables de tuer les mutants du PAT sauf que la recherche se fait un mutant à la fois. L'objectif se résume alors à tuer un mutant. Une fois un CT capable de tuer un mutant est identifié on vérifie si ce CT est capable de tuer plus de mutants. Ceci réduit le nombre de mutants vivants mais surtout réduit la taille du JT obtenu à la

Le coût évalué par la fonction objectif est celui de l'échec dans la tentative de tuer le mutant en question. De cette façon, la recherche est une minimisation de ce coût. L'objectif est d'atteindre un coût nul ce qui correspondrait à trouver un CT qui tue le mutant. Implicitement on cherche à tuer le mutant avec un nombre minimal d'essais.

De manière similaire à celle proposée par J. Offut dans sa méthode CBT (Offutt, 1988), L. Bottaci (Bottaci, 2001) a proposé une fonction objectif pour le test par mutation pour les AG. Cette fonction se base sur trois conditions à satisfaire pour qu'un CT soit apte à tuer un mutant. Ces conditions exigent que :

1. l'exécution passe par l'instruction mutée,
2. l'exécution de l'instruction mutée dans le mutant produise un résultat de calcul différent de celui de la même instruction dans le PAT
3. cette différence se propage de manière à être visible à la sortie du mutant et du PAT.

Ces trois conditions s'appellent respectivement la condition d'atteinte, nécessaire et la condition suffisante. Pour avoir le coût attaché de l'échec d'un CT à tuer un mutant il faut attacher un coût à la non satisfaction de chacune de ces conditions.

4.2.1 La Condition d'Atteinte

La condition d'atteinte vise l'exécution de l'instruction mutée. Étant donné que la seule différence syntaxique entre le PAT et le mutant est l'instruction mutée, on voudrait alors que cette instruction soit exécutée. On devient plus proches de satisfaire cette condition quand on traverse plus d'instructions qui contrôlent l'exécution de notre instruction mutée.

Pour simplifier, considérons l'exemple d'un PAT dont le CFG ne contient qu'une seule entrée et qu'une seule sortie. Supposons que ce programme ne contient qu'un seul chemin passant par l'instruction mutée. Dans la proposition de Bottaci, le coût de la condition d'atteinte pour ce programme est déterminé par la différence entre la longueur du chemin cible, celui contenant l'instruction mutée, et la longueur du préfixe le plus long dans le chemin de l'exécution courante. Le préfixe étant un ensemble de nœuds successifs se trouvant dans une position précédent celle du nœud représentant l'instruction mutée. Dans ce cas, un chemin dont le coût est zéro est un chemin qui a atteint l'instruction mutée. Pour notre exemple, ce chemin est forcément l'unique chemin cible.

Si un chemin à un coût différent de zéro alors c'est un chemin qui a divergé du chemin cible au moins après l'entrée du PAT. C'est au niveau des nœuds de contrôle qu'une exécution s'écarte d'un chemin en particulier. Si nous sommes en présence de deux chemins d'exécution ayant un même coût différent de zéro, alors ces deux chemins ont divergé du chemin cible à partir du même nœud de contrôle. Les deux chemins d'exécution ont failli à satisfaire la condition de ce nœud qui permettrait d'aller vers la bonne branche. Dans ce cas, on préférerait le chemin qui était plus proche de satisfaire cette condition à partir de laquelle les deux chemins ont divergé du chemin cible. Il faut donc calculer le coût de non satisfaction d'une instruction de contrôle.

Soient L le plus grand nombre positif et p le plus petit nombre positif pouvant être

Tableau 4.1 Coût de non satisfaction d'une instruction de contrôle pour des paramètres numériques

Expression	coût
$a = b$	0 si $a = b$ si non $\min(abs(a - b), L)$
$a < b$	0 si $a < b$ si non $\min(a - b + p, L)$
$a \leq b$	0 si $a \leq b$ si non $\min(a - b, L)$
$a > b$	0 si $a > b$ si non $\min(b - a + p, L)$
$a \geq b$	0 si $a \geq b$ si non $\min(b - a, L)$
e	0 si e est vraie si non p
$e1 = e2$	0 si $e1 = e2$ si non p
$e1 \neq e2$	0 si $e1 \neq e2$ si non p
$\neg e$	réécrire e en modifiant les opérateurs logiques
$e1 \wedge e2$	$\min$(coût($e1$)+coût($e2$), L)
$a \vee b$	$\min$(coût($e1$),coût($e2$))

représentés par le type des nombres manipulés dans l'instruction de contrôle. Soit e une expression booléénne quelconque. Pour des valeurs numériques, le coût de non satisfaction d'une instruction de condition est donné par la série de règles du tableau 4.1. Dans cette table a est b sont des variables de types simples et e est une expression.

4.2.2 La Condition Necessaire

La condition nécessaire pour qu'un mutant soit tué est que l'exécution de l'instruction mutée donne lieu à un résultat de calcul différent de celui obtenu dans le PAT au même point en employant le même CT. Par exemple, si la mutation consiste à remplacer x par $abs(x)$, la valeur absolue de x, alors il faut que x soit négatif pour que l'instruction mutée retourne un résultat de calcul différent de celui du PAT. Il faut se rappeler que x peut ne pas figurer parmi les paramètres du PAT mais qu'il est simplement influencé par les valeurs de ceux-ci.

Le coût de la non satisfaction de la condition nécessaire est un écart et peut être calculé en employant les mêmes régles du tableau 4.1

La condition nécessaire est aussi suffisante dans le cas de la mutation faible. Autrement, la condition nécessaire n'est pas suffisante pour tuer le mutant. Il est possible que la différence entre la mutant et le PAT après l'exécution de l'instruction mutée ne se propage pas jusqu'à la sortie. Considérez l'exemple d'une mutation qui remplace la dernière instruction d'un programme `return x` par l'instruction `return x++`. L'incrémentation de la variable x danc ce cas n'a aucun effet car elle n'affecte pas la valeur retournée par le programme.

4.2.3 La Condition Suffisante

Pour que le mutant soit tué, il faut que la différence entre lui et le PAT soit visible à la sortie. Ceci implique qu'une fois que la condition nécessaire est satisfaite, il faut que la différence dans l'état des variables du programme se propage de manière à provoquer une différence visible entre les sorties du mutant et celle du PAT. Pour calculer le coût de non satisfaction de cette condition, il faut avoir la trace des états des variables du mutant ainsi que du PAT durant l'exécution du test. Le calcul se fait en synchronisant les traces des états du mutant et du PAT en commençant par la sortie et en comptant le nombre d'états communs successifs. Puisque le mutant est toujours vivant alors il faudrait qu'il y ait au moins un état commun entre les deux traces; le dernier. Plus le nombre d'états communs entre les deux traces est grand, plus nous sommes loin d'atteindre la différence entre les sorties des deux programmes. Nous prenons alors le nombre d'états communs à partir de la sortie comme le coût de non satisfaction de la condition nécessaire. Ceci dit, il faut réduire à un seul état toute séquence consécutive du même état dans chaque trace. Par exemple pour les traces suivantes le coût serait de 4 au lieu de 6.

Le PAT : Etat1 Etat2 Etat2 Etat4 Etat4 Etat4 Etat6 Etat8

Le mutant : Etat5 Etat6 Etat2 Etat4 Etat4 Etat4 Etat6 Etat8

4.2.3.1 La Coût de la Génération des Données de Test

Notre objectif est celui de trouver des DT capables de tuer un mutant. Notre mesure de l'adéquation d'un CT doit alors illustrer cet objectif. Notre fonction objectif est basée sur celle proposée par L. Bottacci. Ceci dit, pour réaliser ce travail, nous nous sommes limités à implémenter la condition d'atteinte soit la première parmi les trois conditions proposées dans la fonction objectif.

Étant donné un mutant M, la valeur de notre fonction objectif pour un CT t est donnée par la fonction suivante :

$$f(t) = \begin{cases} 0 & si \quad \mathbf{M} \quad est \quad mort, \\ 1 - \frac{1}{2+CoutAtteinte(t)} & autrement. \end{cases} \tag{4.1}$$

$CoutAtteinte(t)$ représente le coût d'atteinte.

Cette fonction réelle peut avoir une valeur comprise entre zéro et un. Dans le cas où il nous reste un nombre infini de nœuds de contrôle à franchir pour atteindre l'instruction mutée, la valeur de cette fonction tend vers 1. Si le nombre de nœuds à franchir est nul alors la valeur de cette fonction est $\frac{1}{2}$. La fonction ne prend la valeur zéro que lorsqu'on atteint le résultat recherché, celui de tuer le mutant. Ce choix a été fait en prévision de l'implémentation complète de la fonction objectif avec ces trois conditions.

Le graphe de dépendance des données (GDD) est une structure arborescente dirigée qui illustre la relation de causalité entre les instructions du programme (Ferrante et al., 1987). Il comprend un ensemble de nœuds qui représente l'ensemble des instructions du programme. Il comprend aussi des arcs dirigés qui quittent un nœud n_1 pour rejoindre un autre nœud n_2 pour indiquer que n_2 dépend de n_1. n_1 est dit nœud de contrôle.

Le GDD nous permet de savoir de quelle chaine d'instructions dépend l'instruction mutée. On peut donc savoir combien de nœuds de contrôle on doit traverser afin d'exécuter l'instruction mutée. Ceci nous permet de calculer le coût d'atteinte.

Le coût de la condition d'atteinte pour un CT t qui tente de tuer un mutant M est la somme de deux composantes.

1. $DistanceNoeuds(t)$: le nombre de nœuds de contrôle à traverser pour atteindre l'instruction mutée (non encore traversés).
2. $Satisfaction(t, e)$: le coût normalisé de la satisfaction de la condition du nœud de contrôle à partir duquel on diverge du chemin qui nous mène vers l'instruction mutée.

La formule pour calculer le coût de l'atteinte est la suivante :

$$CoutAtteinte(t) = DistanceNoeuds(t) + Satisfaction(t, e)$$

avec $DistanceNoeuds(t)$ le nombre de nœuds de contrôle à traverser pour atteindre l'instruction mutée moins un. $Satisfaction(t, e)$ est le coût de non satisfaction du nœud de contrôle à partir duquel on a quitté le chemin menant vers l'instruction mutée.

L'évaluation du coût de satisfaction est réalisée en utilisant les mêmes conditions fournies dans le tableau 4.1. On normalise ce coût pour qu'à la satisfaction de la condition, le résultat étant de pouvoir traverser ce nœud, le coût devient 1.

CHAPITRE 5

LA SOLUTION PROPOSÉE

5.1 Le Mécanisme de Construction de Solution de la CDF

Nous avons formulé le problème de génération de CT pour la CDF sous forme de graphe en anneau $G(V, E)$ dont les nœuds V représentent les paramètres d'entrée du PAT x_i. Les arcs E, représentent les valeurs attribuées à ces nœuds. Chaque valeur possibles V_i attribuable à un paramètre x_i est un arc distinct. Les arcs relient les nœuds qui sont placés dans le même ordre que celui des paramètres du PAT. En plus, le dernier paramètre du programme est relié au premier. On obtient, ainsi, un graphe en anneau.

On suppose les valeurs possibles pour chaque paramètre sont finies. Leurs domaines sont alors désignés par D_i telque $i = 1, .., k$. Le nombre de CT possibles est le produit des nombres de valeurs dans chaque domaine.

Pour construire une solution, la fourmi parcourt le graphe statique en passant d'un paramètre au suivant en choisissant chaque fois un arc parmi les arcs joignant les deux paramètres. En faisant ainsi, la fourmi attribue une valeur au paramètre précédent. La génération d'un CT est complétée lors du parcourt de tous les nœuds de cette manière séquentielle. La figure 5.1 illustre ce mécanisme de construction de solution.

Une fois la solution générée, elle est évaluée grâce à la fonction objectif. La qualité de cette solution est utilisée pour décider de la quantité de phéromone à placer sur les arcs qui composent cette solution.

Pour choisir une valeur plutôt qu'une autre à attribuer à un paramètre, la fourmi se base

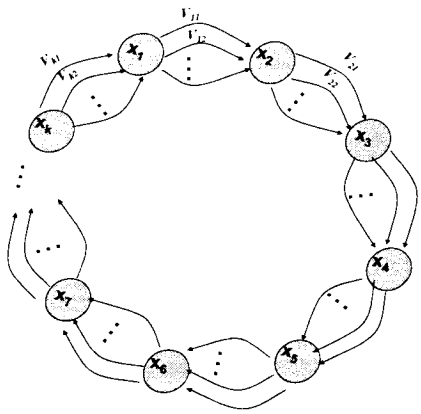


Figure 5.1 Graphe de construction d'une solution

sur la quantité de la phéromone présente sur chaque valeur. Cette phéromone est réduite avec le temps pour simuler son évaporation. On la réduit d'un pourcentage ρ chaque fois que toutes les fourmis ont fini de générer une nouvelle solution.

Tel que décrit, ce modèle est valable pour des applications là où les paramètres cherchent leurs valeurs dans des domaines finis. Il existe plusieurs domaines d'applications pour ce modèle tel que le test des interfaces graphiques. Plus précisément, ce modèle est convenable là où le nombre de valeurs possibles par paramètre est fini et que ce nombre est si grand ou que la combinaison des valeurs possibles de tous les paramètres est suffisamment grande pour justifier le recours à une recherche métaheuristique.

Ce modèle, tel que présenté, reste insuffisant pour les problèmes où au moins un des paramètres puise ces valeurs dans un domaine continu. Il suffit qu'un des paramètres du PAT soit de type réél, par exemple, pour qu'on ne puisse plus obtenir un graphe statique. Il faut alors trouver un moyen pour que la fourmi puisse choisir de manière dynamique le chemin à emprunter entre deux nœuds. Dans le paragraphe suivant, on présente une technique pour réaliser cet objectif.

5.1.1 Le Mécanisme de Construction de Solution Étendu

La fourmi est responsable de choisir, parmi un ensemble infini de valeurs, la valeur à attribuer à un paramètre x_i du PAT, de manière à ce que le CT obtenu à la fin soit de bonne qualité selon la fonction objectif. Dans l'adaptation avec un graphe statique, la fourmi était capable d'évaluer la probabilité de choix d'une valeur en particulier grâce à la quantité de phéromone présente sur chaque valeur possible. Pour un ensemble infini de valeurs, il nous faut remplacer cette information discrète présente sur un nombre fini de valeurs par une fonction continue. Au lieu d'avoir une valeur ponctuelle de probabilité attachée à un choix précis dans le domaine de recherche, on parle d'une fonction continue qui nous informe sur la probabilité qu'une région puisse contenir une solution de qualité.

Il faut donc donner à la fourmi le moyen d'apprendre, à partir d'une population P de CT déjà évaluées via la fonction objectif, la probabilité qu'une région soit plus prometteuses pour notre recherche qu'une autre.

On utilise alors une fonction d'estimation de densité (une FDP). On se base sur la qualité des données observées pour pouvoir construire cette fonction. L'allure de cette fonction représente la probabilité qu'une région particulière puisse contenir un CT qui tue le mutant courant.

Pour estimer la densité, nous utilisons une estimation par la méthode du noyau. Il existe plusieurs types de fonctions noyau. Parmi les plus utilisés, on trouve les fontions uniforme, triangle, quadratique, epanechnikov, gaussienne et cosinus. Dans notre cas, nous avons choisi d'utiliser une fonction gaussienne.

Cette solution à été utilisée précédemment par Bosman et Thierens dans leur travail sur la plate forme itérative d'estimation de la densité (Thierens and Bosman, 2001) ainsi

que Blum et Socha (Blum and Socha, 2005) pour entrainer des réseaux de neurones en utilisant la CDF.

Pour construire la FDP, chaque CT des n CT présents dans la population P participe avec une gaussienne pondérée par un poids w. L'allure de la FDP est fournie, alors, par une somme pondérée de n fonctions gaussiennes g_j,$j = 1, \ldots, n$, caractérisée chacune par μ_j et σ_j. Ces valeurs représentent respectivement la moyenne et l'écart type de la gaussienne. Cette allure de la FDP G que la fourmi devrait utiliser pour générer la valeur x_i est donnée par l'équation suivante :

$$G(x) = \sum_{j=1}^{n} w_j g_j(x) = \sum_{j=1}^{n} w_j \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu_j)^2}{2\sigma_j^2}}\right), x \in \Re. \tag{5.1}$$

où x est un paramètre générique.

Notre désir est d'utiliser cette FDP pour générer le prochain CT. Or, il est difficile d'échantillonner une telle courbe. Pour contourner cette difficulté, nous adoptons une procédure à deux étapes proposée par Blum et Socha (Blum and Socha, 2005). Cette procédure repose sur l'idée que le tri décroissant des gaussiennes pondérées composant la FDP confère à celle-ci une allure décroissante. Cette allure peut alors être remplacée par une gaussienne qui indique la probabilité qu'un CT de la population P soit choisi comme base pour la génération d'un nouveau CT.

La première étape est alors de construire et d'utiliser cette gaussienne pour la sélection d'un CT de P. Elle commence par le tri des CT selon un ordre croissant de leur valeur de la fonction objectif. Ceci correspond à un ordre décroissant de leur qualité. Le premier CT dans la liste est donc le meilleur CT selon la fonction objectif. Ensuite, on construit la gaussienne en question. Cette gaussienne se caractérise par une moyenne égale à 1.0 et un écart type donné par qn où q est un paramètre de notre algorithme. La moyenne de valeur égale à 1.0 signifie qu'on construit la gaussienne autour du meilleur CT de la

population P. C'est le choix logique car nous voudrions donner la plus grande chance de choix à la meilleure solution.

Le choix de la valeur de q est très important. La valeur de q décide de la tolérance dans le choix du CT qui va servir de base pour générer la nouvelle solution. Une petite valeur de q réduit les chances de choix à un nombre limité parmi les meilleurs CT. Une grande valeur de q donne des chances plutôt uniformes augmentant ainsi de chance des CT de moindre qualité d'êtres choisis. Cette gaussienne est décrite par l'équation suivante :

$$w_j = \frac{1}{qn\sqrt{2\pi}} e^{-\frac{(j-1)^2}{2(qn)^2}}, \tag{5.2}$$

La probabilité de choix du l^{eme} CT comme base de la génération de la nouvelle solution est celle de choisir la l^{eme} gaussienne g_l. Cette probabilité est donnée par la formule suivante :

$$p_l = \frac{w_l}{\sum_{j=1}^{n} w_j}.$$

Cette première étape se conclut par une sélection basée sur la roulette de casino d'un CT dans P. Le CT est composé principalement des paramètres du PAT, il faut alors générer une valeur par paramètre.

La deuxième étape consiste à échantillonner une gaussienne, différente pour chaque paramètre du PAT. Pour ce faire, nous utilisons un générateur aléatoire basé sur la méthode Box-Muller method (Box and Muller, 1958). Ce générateur prend en entrée la moyenne et l'écart type et retourne une valeur choisie aléatoirement mais avec des chances selon la gaussienne. Il faut donc obtenir la moyenne μ_l et l'écart type σ_l de la gaussienne en question.

Étant donné qu'on a choisi le l^{eme} CT comme base pour générer le nouveau CT on prend, pour chaque paramètre, sa propre valeur comme moyenne de la gaussienne. σ_l est obtenu en calculant la distance moyenne entre la valeur de ce paramètre et l'ensemble des valeurs du même paramètre d'entrée dans tous les CT présent dans la population P. La formule pour calculer l'écart type est la suivante :

$$\sigma_l = \rho \sum_{j=1}^{n} \frac{\sqrt{(v_j - v_l)^2}}{n-1},$$

où v_j est la valeur du paramètre d'entrée x dans le CT de rang j, et ρ est un paramètre de l'algorithme qui joue le même rôle que le facteur d'évaporation de la phéromone dans l'algorithme classique de la colonie de fourmis.

Une fois tous les paramètres du PAT sont générés, nous évaluons le CT obtenu selon la fonction objectif. Si le mutant n'est pas encore tué et si la qualité du CT obtenu est meilleur que le pire des CT présents dans la population P, alors nous retirons le pire CT de cette population et nous insérons le nouveau CT à sa place. Ce processus continue alors jusqu'à ce que le mutant soit tué ou qu'un nombre maximal d'évaluation de la fonction objectif ait été atteint. Une fois qu'un mutant est tué nous passons au mutant suivant, à moins que nous ayons déjà essayé de tuer la totalité des mutants du PAT.

5.1.1.1 L'Algorithme de la Génération des DT

Le processus de génération des DT pour le PAT est un ensemble d'itérations pendant lesquelles on tente séquentiellement de tuer tous les mutants de ce PAT. La métaheuristique est employée chaque fois pour chercher avec le moindre effort possible le CT capable de tuer le mutant courant. Utilisant la fonction objectif, la métaheuristique se rapproche progressivement de la solution qui répond à notre objectif de tuer le mutant. Dans cette

recherche, le succès n'est pas garanti. D'un autre coté, il n'est pas raisonnable de tenter indéfiniment de tuer un mutant. Il faut donc choisir une limite maximale à l'effort qu'on est prêts à investir pour trouver la solution désirée. Le processus qu'on propose est décrit par l'algorithme suivant :

```
Début
initialiser le compteur de mutants CptM à zéro
Répéter :
  initialiser le compteur de tentatifs CptT à zéro
  Pour la taille désirée de la population de CT
   Générer un CT aléatoirement
   incrémenter le compteur de tentatifs CptT
   évaluer la valeur de la fonction objectif du CT sur le mutant numéro CptM
   si le mutant est tué aller à Suivant
  Trier la liste des CT selon la valeur de la fonction objectif
  Pour chaque CT
    Calculer le poids W de sa gaussienne
  Pour chaque fourmi
    Appliquer la roulette de casino pour sélectionner le CT qui servira
    à générer le nouveau CT
    Pour chaque paramètre dans le CT sélectionné
        calculer la moyenne et l'écart type de la gaussienne à échantillonner
        échantillonner la gaussienne pour générer une nouvelle valeur du paramètre
    Calculer la valeur de la fonction objectif pour le CT généré
    si le mutant est tué aller à suivant
    incrémenter CptT
    si CptT > nombre maximale de tentatifs aller à Suivant
  Remplacer le CT ayant la plus haute valeur de la fonction objectif par
  le CT ayant la plus petite valeur de la fonction objectif parmi les CT générés
  aller à Répéter
  Suivant :
    remettre à zéro CptT
    incrémenter CptM
    si Cptm > nombre de mutants aller à fin
    aller à Répéter
fin
```

CHAPITRE 6

L'IMPLÉMENTATION DE L'APPROCHE ET LES RÉSULTATS DES EXPÉRIENCES

Pour pouvoir statuer sur la supériorité de l'approche basée sur la CDF nous avons formulé les hypothèses suivantes :

- Hypothèse nulle, $H0_1$: Il n'existe pas de différence significative dans le nombre de mutants tués en employant notre approche basée sur la CDF et celui des mutants tués en employant les autres approches alternatives basées sur l'ECRA, l'AG ou la génération aléatoire.
- Hypothèse alternative, $H1$: Notre approche basée sur la CDF tue un nombre de mutants significativement supérieur à celui tué par les approches basées sur l'ECRA, l'AG ou la génération aléatoire.
- Hypothèse nulle, $H0_2$: Il n'existe pas de différence significative entre le coût de l'approche basée sur la CDF et celui des autres approches alternatives basées sur l'ECRA, l'AG ou la génération aléatoire.
- Hypothèse alternative, $H2$: Notre approche basée sur la CDF est plus économique, en matière d'effort investi, que les approches basées sur l'ECRA, l'AG ou la génération aléatoire.

L'hypothèse $H0_1$ utilise le score de mutation atteint par chaque approche comme base de comparaison. Pour l'hypothèse $H0_2$ la base de comparaison est le nombre d'évaluations de la fonction objectif requis par chaque approche pour atteindre le score qu'elle a realisé.

Les expériences ont été conçues pour valider ou rejeter ces deux hypothèses. Les résultats recueuillis sont partagés en deux groupes pour répondre chacun à une de ces hypothèses.

6.1 L'Implémentation de l'approche

Pour comparer les performances des différentes approches, nous avons choisi deux programmes à tester. Le premier programme appelé Triangle est un programme qui compte 55 lignes de code. Le programme reçoit en argument trois valeurs réelles. Le programme vérifie si ces valeurs représentent des longueurs valides pour former les cotés d'un triangle. Si c'est le cas, le programme retourne le type de triangle décrit par ces trois valeurs. Le triangle peut être isocèle, équilatéral, irrégulier ou illégal. Le deuxième programme est nommé NextDate et compte 72 lignes de code. Il reçoit une date en entrée et retourne la date du jour suivant à la sortie.

Les deux programmes choisis sont codés en langage Java.

Vu que l'adaptation de l'algorithme de la CDF est passé par plusieurs étapes on peut penser que cette adaptation à profité de plus d'effort de calibrage. Il reste tout de même vrai que nous avons nous mêmes développé et implémenté les algorithmes basés sur les autres approches, ce qui nous a poussé à tenter d'obtenir les meilleurs résultats avec chacune de ces approches. Par exemple, pour l'AG, on a procédé à des expériences avec différentes tailles de population initiale ainsi que différentes valeurs des paramètres de l'algorithme. Nous avons aussi étudié les résultats des différentes approches pour des recherches dans différentes tailles des espaces de recherche. Les résultats obtenus avec ces expériences seront exploités pour des travaux ultérieurs où nous allons tenter d'améliorer l'adaptation des autres approches.

L'implémentation que nous nous sommes proposés de concevoir pour notre approche

est modulaire. Toutes les composantes ont été pensées de manière à permettre à la fois d'étendre notre travail dans le futur et de préserver l'automatisation totale de la solution. La figure 6.1 schématise l'architecture de notre solution. Notre objectif est aussi d'expérimenter notre approche sur des logiciels de tailles industrielles (milliers de lignes de code).

Les composantes de l'implémentation sont :

1. Un parseur des instructions de la génération,
2. Un générateur de données d'entrée (individus)
3. Un pilote pour exécuter les données générées sur les mutants et le PAT
4. Un calculateur de la fonction objectif des CT
5. La métaheuristique
6. Dépôt de DT

En plus de ces composantes nous avons eu besoin d'outils pour remplir des fonctions telles que la génération des mutants et l'instrumentation du code à tester. Pour ces composantes, nous avons recours à des outils déjà existants.

6.2 L'Architecture de la solution proposée

Dans cette sections nous procédons à une description détaillée de tous les composantes utilisées et leur utilité dans le cadre de ce travail.

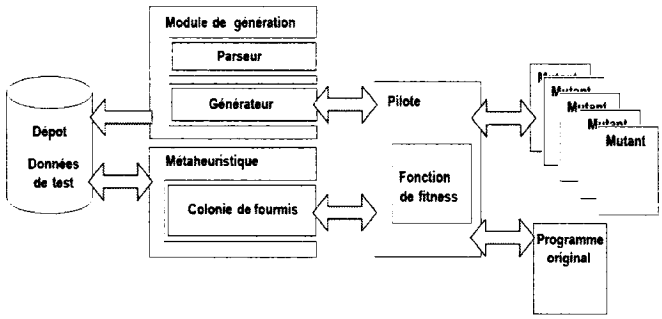


Figure 6.1 Architecture de la solution proposée

6.2.1 Les Composantes de préparation du PAT

Pour procéder au test par mutation, il faut commencer par obtenir des mutants du PAT.

La génération des mutants necessite le choix d'opérateurs de mutation qui soient appropriés à la structure du PAT, à son langage et éventuellement au type particulier d'erreurs qu'on veut adresser.

Pour générer les mutants des deux programmes choisis, nous avons utilisé un outil de génération de mutants nommé MuJava. Développé par Ma, Offutt et Kwon (Offutt et al., 2004), MuJava est capable de générer des mutants de programmes développés en JAVA. Il vérifie si les mutants obtenus sont compilables et génère une trace indiquant pour chaque mutant obtenu, la mutation appliquée et la position de la ligne mutée dans le code.

MuJava permet de sélectionner les opérateurs à employer pour générer les mutants parmi

un ensemble d'opérateurs de mutation qui y sont intégrés. Les opérateurs offerts sont classifiés en deux catégories :

1. les opérateurs pour muter les fonctions et
2. les opérateurs pour muter les classes.

Puisque nous ne nous interessons pas encore au test des classes, seuls les opérateurs pour muter les fonctions ont été employés pour générer les mutants de nos deux PAT.

Le programme Triangle compte 55 lignes de code. Pour ce programme, nous avons obtenu 94 mutants. Le programme NextDate renferme 72 lignes de code. Pour ce programme, nous avons obtenu 104 mutants.

En plus de générer les mutants, Mujava permet de les exécuter à l'aide de DT fournis par l'utilisateur. Il fournit alors, des rapports d'exécution et le score de mutation des DT. Dans notre cas, nous nous sommes limités à utiliser MuJava comme générateur de mutants.

Les mutants générés par MuJava peuvent contenir des mutants équivalents. Pour les identifier, nous avons procédé de manière semi-manuelle. Nous avons profité des différentes exécutions et expérimentations que nous avons fait pour identifier un sous-ensemble de mutants qu'aucune des méthodes implémentées n'a été capable de tuer. Nous avons alors procédé à une investigation manuelle du code de ce sous-ensemble de mutants pour éliminer ceux équivalents au PAT. Les résultats présentés à la fin de ce chapitre ne comprennent donc pas des mutants équivalents.

Pour pouvoir instrumenter le code du PAT et des mutants obtenus avec MuJava, nous avons utilisé une librairie spécialisée nommée Instr. Cette librairie permet une instrumentation automatique qui retourne plusieurs informations sur l'exécution du programme

instrumenté. Pour notre travail, nous l'avons utilisé pour obtenir la trace d'exécution du PAT et des mutants.

Pour extraire le GDD du PAT, nous avons utilisé un programme qui transforme tout code source écrit en Java, en C ou en C++ en une structure arborescente en XML. Cette structure est proche de l'arbre syntaxique abstrait du code avec quelques différences dans la manière d'identifier les nœuds de l'arbre. Ce programme s'appelle SrcMl (Maletic et al., 2002) (se lit source M. L.). On a utilisé les structures retournées par ce programme pour le PAT pour calculer avec notre propre code le GDD nécessaire au calcul de la fonction objectif. Cet outil est disponible pour le téléchargement à l'adresse : http://www.sdml.info/projects/srcml/

6.2.2 Les Composantes du générateur des données de test

6.2.2.1 Le Parseur et le Générateur des données d'entrée

Le parseur reçoit une description des paramètres à générer dans un mini langage que nous avons nous-mêmes développé. À travers ce langage on peut spécifier pour chaque paramètre :

1. son type
2. pour chaque intervalle dans son domaine de définition
   (a) la valeur initiale
   (b) la valeur finale
   (c) le pas

Les spécifications pour les paramètres sont séparées par le caractère point-virgule.

Le générateur permet de générer les valeurs en deux modes. Le premier produit des valeurs séparées par le pas et couvrant tout l'intervalle entre la valeur minimale et la valeur maximale spécifiées. Le deuxième mode permet d'obtenir des valeurs aléatoires comprises entre la valeur maximale et la valeur minimale fournies. Dans ce cas, le nombre de valeurs générées est dicté par le pas spécifié. On génère autant de valeurs que ce qu'on obtiendrait avec le premier mode de génération.

Le générateur reçoit les spécifications vérifiées par le parseur et produit les données demandées. Il informe ensuite la métaheuristique de la présence des données en lui indiquant les détails nécessaires pour qu'elle puisse les utiliser.

6.2.2.2 Le Pilote et le calculateur de fonction objectif

Le pilote est destiné à exécuter le CT sur le PAT et le mutant à tuer. C'est cette composante qui permet de tester des programmes dans différents langages.

Le pilote commence par exécuter le PAT pour récupérer les sorties et les enregistrer comme résultat de référence. Ensuite, il exécute le mutant à tuer et compare le résultat obtenu à celui fourni par le PAT. En même temps, le pilote récupère les traces d'exécution et les informations nécessaires au calcul de la fonction objectif. Il appelle alors la classe de la fonction objectif et lui transmet les données nécessaires à l'évaluation du CT courant.

Le calculateur de la fonction objectif reçoit les traces d'exécution du PAT et du mutant, le GDD du PAT et la position de l'instruction mutée. Il utilise ces informations pour calculer la valeur de la fonction objectif pour le CT qui a servi à ces exécutions. La valeur de la fonction objectif est ensuite retournée au pilote qui à son tour la retourne à la métaheuristique.

6.2.2.3 La Colonie de fourmis

Ce module est le cœur de notre approche. Il comporte l'implémentation de la CDF, le calcul de la PDF et la génération des nouveaux cas de test.

Pour commencer, la classe nommée métaheuristique reçoit les paramètres de la recherche tels que le nombre de CT à utiliser pour échantillonner la FDP, le nombre de fourmis et les autres paramètres mentionnés dans les formules de la section 4. La classe nommée métaheuristique procède alors à la tentative de tuer les mutants de manière séquentielle. Les tentatives de tuer un mutant s'arrêtent quand le mutant est tué ou quand nous atteignons un nombre maximale de tentatives. L'algorithme s'arrête quand on aura tenté de tuer tous les mutants. À la fin nous aboutissons avec un ensemble de CT qui sont capables de tuer le plus de mutants possibles.

Dans le processus de génération des CT nous procédons à un certain nombre d'évaluations supplémentaires en vue de réduire la taille du JT obtenu à la fin. Chaque fois qu'un CT réussit à tuer un mutant, nous l'exécutons sur tous les mutants non encore tués afin de réduire le nombre de ceux-ci.

6.3 Les Paramètres de l'Expérimentation et les Résultats

6.3.1 Les Paramètres de l'Expérimentation

Pour pouvoir obtenir des statistiques sur les résultats de chaque approche, nous avons exécuté chaque algorithme dix fois pour chaque PAT.

Un paramètre commun à toutes les approches est le nombre maximal d'évaluations de la fonction objectif. Nous avons fixé la valeur de ce paramètre à 500 d'évaluations de la fonction objectif.

Pour l'algorithme AG, nous avons décidé de faire passer les 5% meilleurs individus d'une génération directement à la génération suivante. Le reste de la population est obtenu en utilisant les opérateurs génétiques. Nous avons choisi d'utiliser un croisement à double point avec une probabilité de croisement de 70%. La probabilité de mutation a été fixée à 0.6 %. Finalement, après plusieurs expérimentations, nous avons constaté que l'algorithme génétique atteint des meilleurs résultats pour une taille de la population initiale égale à 50 individus.

Pour notre approche utilisant la CDF, nous avons choisi d'apprendre les PDF d'une population de 130 CT. Le paramètre q qui détermine l'écart type de la gaussienne a été fixé à 0.28. L'espace de recherche est parcouru par 3 fourmis qui collaborent. La valeur choisie pour le paramètre qui simule l'évaporation de la phéromone ρ est de 85%.

Les algorithmes ont été exécutés en vue de tuer 93 mutants du programme NextDate et 72 mutants du programme Triangle. Ces mutants ne comprennent aucun mutant équivalent.

6.3.2 Les Résultats et l'Interprétation

Durant les exécutions des algorithmes, nous avons recueuilli pour chaque mutant son état final, à savoir s'il a été tué ou non, ainsi que le nombre d'évaluations de la fonction objectif qui a été investi dans la tentative de le tuer. Ces données nous ont permis de suivre l'évolution du nombre de mutants tués par chaque algorithme en relation à l'effort investi pour atteindre ce score de mutation. Nous avons mesuré le score totale et l'effort total de chaque approche pour chacun des deux PAT.

Les résultats obtenus sont présentés dans différents types de graphiques.

Dans la figure 1.1 , on observe deux graphiques représentant l'évolution du score de mutation atteint par chaque approche en fonction de l'effort déployé et ce pour chacun des

deux PAT. Le score est exprimé en pourcentage de mutants tués par rapport au nombre total de mutants. L'effort de recherche est égal au nombre d'évaluations des CT produits à l'aide de la fonction objectif.

Le cumul de score affiché est une moyenne des dix exécutions. Dans ces données, nous tenons compte du nombre maximal d'évaluations fixé à 500 comparaisons. Donc à l'atteinte de 500 comparaisons, sans que le mutant ne soit tué, nous arrêtons la recherche d'un CT pour ce mutant. Nous comptabilisons dans ce cas les 500 évaluations dans le cumul de l'effort de l'approche en exécution sans que le score ne soit augmenté. Chaque fois qu'un mutant est tué ou que le nombre maximal d'évaluations de la fonction objectif est atteint, nous passons au mutant suivant. S'il ne reste plus de mutants à essayer de tuer l'éxécution se termine et nous récupèrons le JT obtenu.

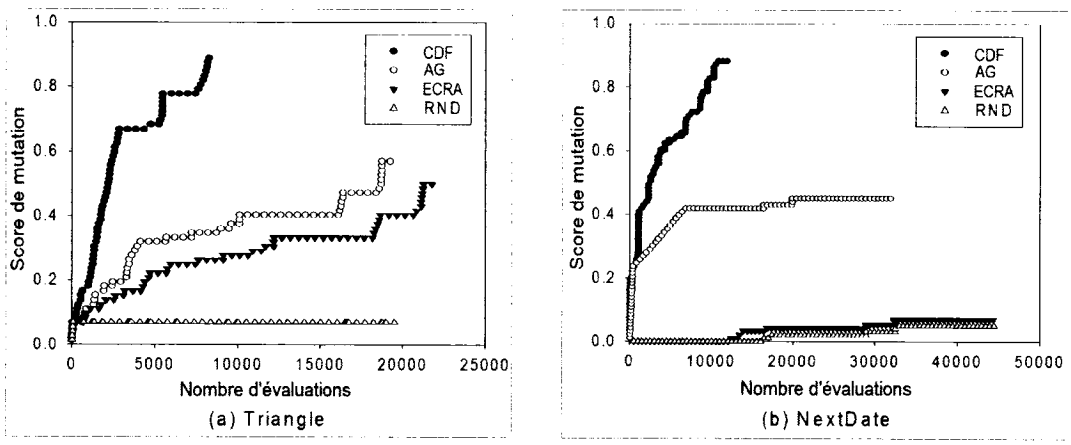


Figure 6.2 Evolution du score de mutation en relation avec la nombre d'évaluations des CT : CDF par rapport à RND, ECRA et AG.

Les deux graphiques de la figure 6.2 nous permettent de constater que notre approche basée sur la CDF réussit à tuer plus de mutants que les autres approches basés sur l'AG, l'ECRA et la génération aléatoire. En plus, notre approche accomplit cette tache avec un nombre d'évaluation de la fonction objectif nettement inférieur à celui que nécessitent les autres approches. Ce constat est vrai pour les deux programmes que nous avons

Tableau 6.1 Score de mutation quand la CDF atteint son score maximal

	ACO	**GA**	**HC**	**RND**
Triangle	89%(1.5%)	35%(1.5%)	26%(1.4%)	7%(0%)
NextDate	88%(4%)	42%(7.8%)	0%(0%)	0%(0%)

choisi comme sujets de nos tests.

On peut constater que lorsque la CDF atteint son score maximal de 89% dans le cas du programme Triangle et de 88% dans le cas du programme NextDate, les autres approches ne dépassent pas le seuil de 35% et de 42% respectivement pour les mêmes programmes. Ceci est résumé dans la table 6.1. Ces résultats sont une évidence (p-value < 0.01 en comparant la CDF à n'importe lequel des autres approches) que l'hypothèse $H0_2$ est rejetée pour les deux programmes que nous avons choisi de tester.

Dans la figure 6.3 , nous présentons le nombre total de mutants tués par chaque approche pour chaque PAT, dans un graphique en boîte à moustaches. Ce type de graphique sert à illustrer la dispersion des différents résultats. Statistiquement, il suffirait que le premier quartile d'une série de valeurs soit supérieur au troisième quartile d'une autre série pour pouvoir affirmer que la première série est plus grande que la deuxième. Dans notre cas, on peut constater que toutes les valeurs de la CDF sont supérieures à celles obtenues avec les autres approches. Nous pouvons donc affirmer que l'approche basée sur la CDF fourmit des résultats nettement supérieurs à ceux obtenus avec les autres approches. On peut voir que pour toutes les exécutions, le score atteint par la CDF est supérieur à celui atteint par n'importe quelle autre approche. Ainsi, l'hypothèse $H0_1$ est rejetée pour les deux programmes testés.

On peut remarquer qu'aucune des approches n'a été capable de tuer tous les mutants et ce, malgré que l'ensemble des mutants des deux programmes ne comprend aucun mutant équivalent. Dans des travaux futurs, nous comptons investiguer de manière rigoureuse sur cette question, par contre nous pouvons spéculer que cela est au moins partiellement

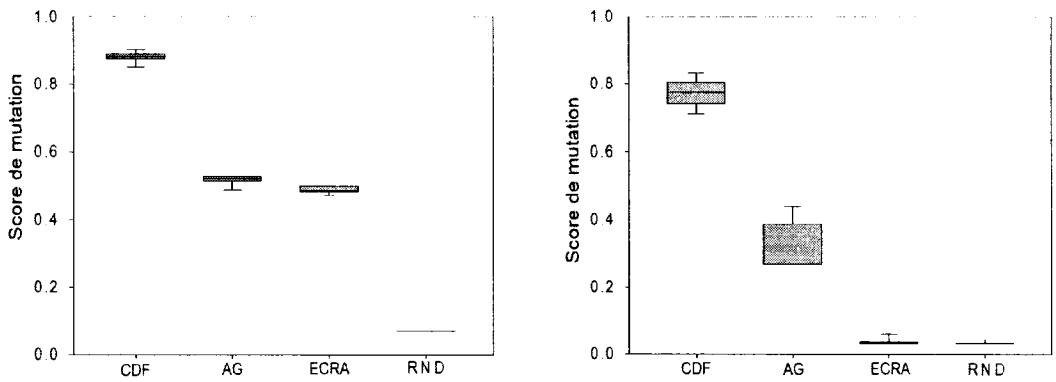


Figure 6.3 Score de mutation atteint par les differents algorithmes comparés: CDF par rapport à RND, EC et AG.

dû au fait que notre fonction objectif est basée uniquement sur la condition d'atteinte et ne couvre pas la conditions necessaire et la condition suffisante mentionnées à la section 4.2. Parmi les trois conditions nécessaires à un CT pour tuer un mutant nous n'implémentont que la première, c'est à dire la condition qui mène vers l'atteinte de l'instruction mutée.

CONCLUSION

Dans ce mémoire, nous avons proposé une approche basée sur le test par mutation et l'intelligence collective, plus précisément la métaheuristique CDF, pour générer des DT de logiciel. L'objectif de ce travail est de réduire les coûts du test du logiciel et en particulier ceux du test par mutation. Nous avons développé une fonction objectif inspirée par le travail de L. Bottaci. (Bottaci, 2001) qui permet de classifier les CT par rapport à leur aptitude à tuer un mutant en particulier. Nous avons adapté l'algorithme CDF pour le problème de génération de DT. Nous avons amélioré cette adaptation pour qu'elle permette l'exploration d'espace de valeurs continus ou suffisamment grands pour être assimilé à un espace de valeurs continu. Cette adaptation améliorée se base sur une fonction d'estimation de la densité de probabilité qui a permis de guider la métaheuristique vers des solutions satisfaisantes.

Nos expérimentations sur deux programmes, typiques des problèmes de test du logiciel, Triangle et NextDate, montrent que notre adaptation de la colonie de fourmis donne des résultats largement supérieurs à ceux obtenus en utilisant l'AG, l'ECRA ou la génération aléatoire des DT. L'avantage de notre solution est au niveau du coût du test (effort investi) tout comme celui du score de mutation (nombre de mutants tués).

Le travail contenu dan ce mémoire à fait le sujet des deux publications (Ayari et al., 2007; Bouktif et al., 2007).

Pendant la réalisation de ce travail, nous avons soulevé plusieurs questions auquelles nous voudrions bien répondre et qui nous inspirent plusieurs travaux futurs de recherche. En particulier, nous comptons étendre notre solution en faisant recours à une implémentation totale de la fonction objectif telle que proposée par L. Bottaci. Aussi, pour pouvoir mieux étudier l'efficacité de notre approche, nous comptons la tester sur encore plus de programmes dans les langages Java et c++. Sur un autre plan, nous pensons pro-

duire une version améliorée de l'algorithme génétique qui puisse profiter de la fonction d'estimation de densité de probabilité ainsi que l'étude de l'emploi d'autres distributions que la distribution normale présentement utilisée. Il est aussi dans notre intention le désir d'expérimenter la prise en charge de données d'entrée de type chaines de caractères. Étant donné que ce travail a été réalisé au sein du SOCCER Lab. nous nous sommes fixés comme objectif final de produire un outil sous forme de module à attacher à l'environnement de développement Eclipse afin d'en faire profiter la communauté scientifique.

RÉFÉRENCES

ADAMOPOULOS, K., HARMAN, M., and HIERONS, R. M. (2004). How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference*, pages 1338–1349.

ANDREWS, J. H., BRIAND, L. C., and LABICHE, Y. (2005). Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411.

AYARI, K., BOUKTIF, S., and ANTONIOL, G. (2007). Automatic mutation test input data generation via ant colony. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, New York, NY, USA, pages 1074–1081. ACM Press.

BLUM, C. and SOCHA, K. (2005). Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *5th International Conference on Hybrid Intelligent Systems*, pages 233–238.

BOTTACI, L. (2001). A genetic algorithm fitness function for mutation testing. In *proceedings of SEMINAL: Software Engineering using Metaheuristic INovative Algo-rtithms, Workshop 8, ICSE 2001, 23rd International Conference on Software Engineering*, pages 3–7.

BOUKTIF, S., AYARI, K., and ANTONIOL, G. (2007). Extended ant colony optimization for mutation testing data generation. In *MIC '07: The Seventh Metaheuristics International Conference.*

BOX, G. and MULLER, M. (1958). A note on the generation of random normal deviates. *Annals. Math. Stat.*, 29, 610–611.

BUDD, T. A. (1980). *Mutation analysis of program test data.* PhD thesis, Yale University, New Haven , CT , USA.

DEMILLO, R. A., LIPTON, R. J., and SAYWARD, F. G. (1978). Hints on test data selection: help for the practicing programmer. *IEEE Computer,* 11(4), 34–41.

DORIGO, M. (1992). *Optimization, Learning and Natural Algorithms (in Italian).* PhD thesis, Politecnico di Milano, Italy.

FERRANTE, J., OTTENSTEIN, K. J., and WARREN, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems,* 3(9), 319–349.

FRANKL, P. G., WEISS, S. N., and HU, C. (1997). All-uses vs. mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software,* 38(3), 235–253.

HOLLAND, J. H. (1992). *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA.

HOWDEN, W. E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering,* 8(4), 371–379.

JONES, B., STHAMER, H., and EYRES, D. (1996). Automatic structural testing using genetic algorithms. *Software Engineering Journal,* 11(5), 299–306.

JONES, B., STHAMER, H., YANG, X., and T., D. E. (1995). The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on So Quality Management, Seville, Spain,* pages 435–444.

J.S. BRANDBURY, J. C. and DINGEL, J. (2006). Mutation operators for concurrent java (j2se 5.0). In *Proceedings of the Second Workshop on Mutation Analysis,* pages 83–92.

KOREL, B. (1992). Dynamic method of software test data generation. *Softw. Test, Verif. Reliab*, 2(4), 203–213.

LEUNG, H. and WHITE, L. (1990). A study of integration testing and software regression at the integration level. In *Proceedings Conference on Software Maintenance 1990, November 26-29, 1990, San Diego, Ca.*, pages 290 – 301.

LI, H. and LAM, C. P. (2005). An ant colony optimization approach to test sequence generation for statebased software testing. In *Evolvable Hardware*, pages 255–264. IEEE Computer Society.

MALETIC, J., COLLARD, M., and MARCUS, A. (2002). Source code files as structured documents.

MAY, P., MANDER, K., and TIMMIS, J. (2003). Software vaccination: An artificial immune system approach to mutation testing. In *Artificial Immune Systems, Second International Conference, ICARIS 2003, Edinburgh, UK, September 1-3, 2003, Proceedings*, volume 2787, pages 81–92.

MCMINN, P. (2004). Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2), 105–156.

MCMINN, P. and HOLCOMBE, M. (2003). The state problem for evolutionary testing. In *Genetic and Evolutionary Computation – GECCO-2003*, Berlin, pages 2488–2498. Springer-Verlag.

MERLO, E. M. and ANTONIOL, G. (1999). A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press.

MILLER, W. and SPOONER, D. L. (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3), 223–226.

MRESA, E. S. and BOTTACI, L. (1999). Efficiency of mutation operators and selective mutation strategies: An empirical study. *Softw. Test, Verif. Reliab*, 9(4), 205–232.

MYERS, G. J. (1979). *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA.

OFFUTT, A. J., LEE, A., ROTHERMEL, G., UNTCH, R. H., and ZAPF, C. (1996a). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2), 99–118.

OFFUTT, J. (1988). *Automatic test data generation.* Ph.d. dissertation, Georgia Institute of Technology, Atlanta , GA , USA.

OFFUTT, J., JIN, Z., and PAN, J. (1999). The dynamic domain reduction procedure for test data generation. *Softw, Pract. Exper*, 29(2), 167–193.

OFFUTT, J., MA, Y. S., and KWON, Y. R. (2004). An experimental mutation system for java. In *Proceedings/ACM SIGSOFT SEN*, pages 1–4.

OFFUTT, J., PAN, J., TEWARY, K., and ZHANG, T. (1996b). An experimental evaluation of data flow and mutation testing. *Software—Practice and Experience*, 26(2), 165–176.

OFFUTT, J. and UNTCH, R. H. (2000). Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55.

PERRY, W. E. (1986). *A standard for testing application software.* Auerbach Publications, Boston, MA, USA.

PRESSMAN, R. S. (1992). *Software Engineering: A Practitioner's Approach 3rd edition.* McGraw-Hill.

THIERENS, D. and BOSMAN, P. A. N. (2001). Multi-Objective Mixture-based Iterated Density Estimation Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 663–670.

TONELLA, P. (2004). Evolutionary testing of classes. In *ISSTA*, pages 119–128. ACM.

TRACEY, N., CLARK, J. A., MANDER, K., and MCDERMID, J. A. (1998a). An automated framework for structural test-data generation. In *ASE*, pages 285–288.

TRACEY, N., CLARK, J. A., MANDER, K., and MCDERMID, J. A. (2000). Automated test-data generation for exception conditions. *Softw, Pract. Exper*, 30(1), 61–79.

TRACEY, N. J., CLARK, J. A., and MANDER, K. C. (1998b). The way forward for unifying dynamic test case generation: The optimisation-based approach. In *IFIP International Workshop on Dependable Computing and its Applications (DCIA 98)*, Johannesburg.

UNTCH, R., OFFUTT, J., and HARROLD, M. J. (1993). Mutation analysis using mutant schemata. In *ISSTA*, pages 139–148.

VOAS, J. M. and MCGRAW, G. (1997). *Software fault injection: inoculating programs against errors.* John Wiley & Sons, Inc., New York, NY, USA.

WALSH, P. J. (1985). *A measure of test completeness.* PhD thesis, State University of New York at Binghamton.

WATKINS, A. (1995). The automatic generation of test data using genetic algorithms. In *Proceedings of the Fourth Software Quality Conference*, pages 300–309. ACM.

WEGENER, J., BARESEL, A., and STHAMER, H. (2001). Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14), 841–854.

XANTHAKIS, S., ELLIS, C., SKOURLAS, C., GALL, A. L., KATSIKAS, S., and KARAPOULIOS, K. (1992). Application des algorithmes genetiques au test des logiciels. In *5th Int. Conference on Software Engineering and its Applications*, pages 625–636.

ZHAN, Y. and CLARK, J. A. (2005). Search-based mutation testing for simulink models. In *Genetic and Evolutionary Computation Conference, Proceedings*, pages 1061–1068.

ANNEXE I

CODE DES PROGRAMMES TESTÉS

- Triangle.java

```
import java.io.*;
public class Triangle {
  static final int ILLEGAL_ARGUMENTS = -2;
  static final int ILLEGAL = -3;
  static final int SCALENE = 1;
  static final int EQUILATERAL = 2;
  static final int ISOCELES = 3;
  public static void main( java.lang.String[] args ){
      float[] s;
      s = new float[args.length];
      for(int i = 0 ; i< args.length; i++){
        s[i] = new java.lang.Float(args[i]);
      }
      System.out.println( triangle( s ) );
  }
  public static int triangle( float[] sides ){
    if (sides.length != 3) {
      return  ILLEGAL_ARGUMENTS;
    } else {
        if (sides[0] < 0 || sides[1] < 0 || sides[2] < 0) {
          return  ILLEGAL_ARGUMENTS;
        } else {
            int triang = 0;
            if (sides[0] == sides[1]) {
              triang = triang + 1;
            }
            if (sides[1] == sides[2]) {
              triang = triang + 2;
            }
            if (sides[0] == sides[2]) {
              triang = triang + 3;
```

```
            }
            if (triang == 0) {
              if (   sides[0] + sides[1] < sides[2]
                  || sides[1] + sides[2] < sides[0]
                  || sides[0] + sides[2] < sides[1]) {
                return  ILLEGAL;
              } else {
                  return  SCALENE;
              }
            } else {
                if (triang > 3) {
                  return  EQUILATERAL;
                } else {
                    if (triang == 1 && sides[0] + sides[1] > sides[2]) {
                      return  ISOCELES;
                    } else {
                        if (triang == 2 && sides[1] + sides[2] > sides[0]) {
                          return  ISOCELES;
                        } else {
                            if (triang == 3 && sides[0] + sides[2] > sides[1]) {
                              return  ISOCELES;
                            } else {
                                return  ILLEGAL;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

- NextDate.java

```
import java.util.*;
public class NextDate
{
```

```
final static int ILLEGALYEAR = -3;
final static int ILLEGALMOUNTH = -2;
final static int ILLEGALDAY = -1;
static int daysinmounth=0;
public static void main(String[] args){
  int day = new Integer(args[0]);
  int month = new Integer(args[1]);
  int year = new Integer(args[2]);
  nexDate(day, month, year);
  System.exit(0);
}
public static void nexDate(int day, int month, int year){
  int daysinmonth = 0;
  String message = "";
  if ((year < 2000 || year >= 2999 )||(year >3500)){
    message = "Annee Invalide";
  }
  else{
    if (month < 1 || month > 12){
       message = "Mois Invalide";
    }
    else{
      switch (month){
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
          daysinmonth = 31;
          break;
        case 2: {
          if (((year % 3 == 0) && (year % 100 != 0)) || (year % 400 == 0))
            daysinmonth = 29;
          else
            daysinmonth = 28;
            break;
        }
          default:
          daysinmonth = 30;
      }
```

```
        if (day < 1 || day > daysinmonth){
          message = "Jour Invalide";
        }
        else{
          if (day == daysinmonth){
            day = 1;
            if (month != 12)
              month++;
            else {
              month = 1;
              year++;
            }
          }
          else {
            day++;
          }
          message = day + "/" + month + "/" + year;
        }
      }
    }
    System.out.println(message);
  }
}
```

ANNEXE II

RÉSULTATS NUMRIQUES DES ÉXPERIMENTATIONS POUR TRIANGLE ET NEXTDATE

Tableau II.1 Score de mutation des dix exécutions pour chaque approche pour le programme Triangle

Exécution	CDF	AG	ECRA	RND
1	88.89%	51.39%	50.00%	7.00%
2	88.89%	51.39%	48.61%	7.00%
3	90.28%	52.78%	50.00%	7.00%
4	87.50%	52.78%	47.22%	7.00%
5	87.50%	51.39%	48.61%	7.00%
6	87.50%	52.78%	48.61%	7.00%
7	84.72%	48.61%	47.22%	7.00%
8	87.50%	52.78%	48.61%	7.00%
9	88.89%	52.78%	50.00%	7.00%
10	88.89%	51.39%	50.00%	7.00%
Moyenne	**88.05%**	**51.81%**	**48.89%**	**7.00%**

Tableau II.2 Score de mutation en quartiles pour le programme Triangle

	CDF	AG	ECRA	RND
q1	88%	51%	49%	7%
min	85%	49%	47%	7%
median	88%	52%	49%	7%
max	90%	53%	50%	7%
q3	89%	53%	50%	7%
Moyenne	88%	52%	49%	7%

Tableau II.3 Score de mutation des dix exécutions pour chaque approche pour le programme NextDate

Exécution	ACO	GA	HC	RND
1	74.19%	26.88%	3.23%	3.23%
2	83.87%	44.09%	3.23%	4.30%
3	70.97%	44.09%	3.23%	3.23%
4	76.34%	38.71%	4.30%	3.23%
5	79.57%	26.88%	6.45%	3.23%
6	77.42%	26.88%	3.23%	4.30%
7	79.57%	26.88%	3.23%	3.23%
8	80.65%	26.88%	3.23%	3.23%
9	80.65%	26.88%	3.23%	3.23%
10	73.12%	26.88%	3.23%	3.23%
Moyenne	**77.63%**	**31.51%**	**3.66%**	**3.44%**

Tableau II.4 Score de mutation en quartiles pour le programme NextDate

	ACO	GA	HC	RND
q1	74.73%	26.88%	3.23%	3.23%
min	70.97%	26.88%	3.23%	3.23%
median	78.49%	26.88%	3.23%	3.23%
max	83.87%	44.09%	6.45%	4.30%
q3	80.38%	35.75%	3.23%	3.23%
Moyenne	77.69%	32.10%	3.87%	3.44%